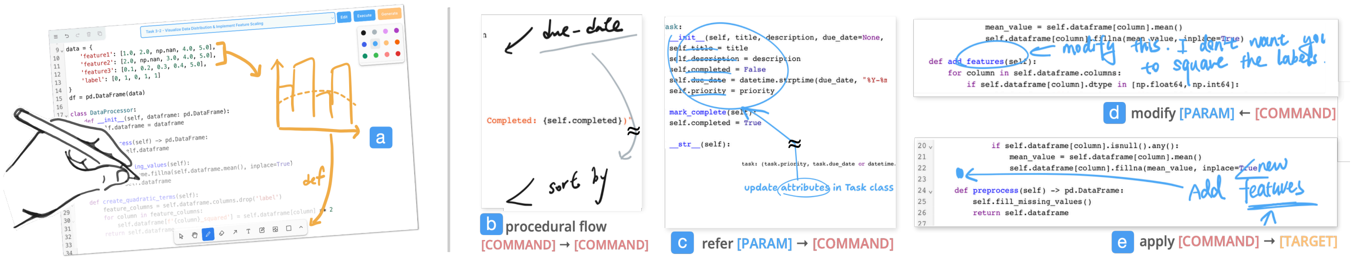
Code Shaping: Iterative Code Editing with Free-form Sketching

Ryan Yen
School of Computer Science,
University of Waterloo
r4yen@uwaterloo.ca

Jian Zhao
School of Computer Science,
University of Waterloo
jianzhao@uwaterloo.ca

Daniel Vogel
School of Computer Science,
University of Waterloo
dvogel@uwaterloo.ca


a
def
due-date
sort by
b procedural flow
[COMMAND] → [COMMAND]
update attributes in Task class
c refer [PARAM] → [COMMAND]
modify this. I don't want you to square the labels.
d modify [PARAM] ← [COMMAND]
Add new features
e apply [COMMAND] → [TARGET]

```
ask:
  __init__(self, title, description, due_date=None,
  self.title = title
  self.description = description
  self.completed = False
  self.due_date = datetime.strptime(due_date, "%Y-%m
  self.priority = priority

  mark_complete(self):
  self.completed = True

  __str__(self):
                    task: (task.priority, task.due_date or datetime.
```

```
            mean_value = self.dataframe[column].mean()
            self.dataframe[column].fillna(mean_value, inplace=True)

    def add_features(self):
        for column in self.dataframe.columns:
            if self.dataframe[column].dtype in [np.float64, np.int64]:
```

```
            if self.dataframe[column].isnull().any():
                mean_value = self.dataframe[column].mean()
                self.dataframe[column].fillna(mean_value, inplace=True

    def preprocess(self) -> pd.DataFrame:
        self.fill_missing_values()
        return self.dataframe
```

Figure 1: (a) A programmer sketches an arrow pointing from data attributes to a drawn bar chart and annotates the code with *def* to generate edited code. Right: Data collected from the user study showing how users employ arrows (⟶) for different purposes, including: command (the intended action of operation), parameter (supplementing the command), and target (the area where the edit should occur); (b) indicating procedural flow between commands; (c) referring to data attributes; (d) modifying a function, with the function as the parameter to supplement the command; (e) applying changes to a target area.

ABSTRACT

We present an initial step towards building a system for programmers to edit code using free-form sketch annotations drawn directly onto editor and output windows. Using a working prototype system as a technical probe, an exploratory study ($N = 6$) examines how programmers sketch to annotate Python code to communicate edits for an AI model to perform. The results reveal personalized workflow strategies and how similar annotations vary in abstractness and intention across different scenarios and users.

KEYWORDS

ink-based sketching, programming interface

1 INTRODUCTION

Many programmers use free-form sketching to externalize ideas to plan high-level structure, workout algorithms, and annotate code. Code annotations in particular serve various purposes, such as enhancing code comprehension [2, 11], communicating with collaborators [5], and planning future edits [9, 10]. However, past practice and previous research mostly treat sketched annotations on code as static externalizations of a programmer's thoughts [7], not as actionable commands to interactively edit code. We propose a sketch-based editing approach where a programmer iteratively draws free-form annotations on and around a code editor to iteratively modify structure, flow, and syntax: a concept we call *code shaping*. For example, to insert a new function to draw charts, the programmer could circle lines of code about data attributes, draw an arrow to a sketch of a graph, then draw an arrow with the word "def" leading back to an insertion point in the code (Figure 1a).

, ,
.

We differentiate the concept of code shaping from another line of research that focuses on converting sketched drawings, such as visualizations or user interfaces on a canvas, into code [4, 12]. These sketches directly represent the final output, without considering the syntactic structure of the code. Instead, our research explores editing code directly through sketches made on the code editor itself. This approach enables programmers to sketch annotations that encapsulate their expectations of how the program should work and connect these sketches with the syntactic code. While recent advances in multi-modal large language models have made this concept more feasible, this approach introduces challenges that need to be understood and addressed. First, the similar annotation could have different meanings across various scenarios and tasks (Figure 1b-e), stemming from the ambiguous nature of sketches [1, 3]. Second, the obscure nature of AI models forces users to guess the reasons for recognition failures and opportunistically change the annotations to make them work. These challenges highlight the need for a system that supports users in iteratively clarifying and modifying both code and annotations.

We conducted an exploratory study with six programmers, using a prototype that transforms free-form sketches on a code editor into actual code edits and reported results in section 4.

2 PROTOTYPE SYSTEM

Our system integrates a code editor within a canvas environment to enable code shaping. The interface supports various free-form sketching tools, including color selectors, pens, erasers, and shapes. A text tool is available for conventional editing. Two-finger panning and zooming navigate the code in the editor to enable sketching at different levels of granularity. A pointer tool can select one or more annotations. Pressing a "Generate" button uses all current annotations, or only selected annotations, as parameters for generating edited code. The system recognizes free-form annotations on the code editor, utilizing GPT-4o to generate corresponding code. We render HTML content from both the code editor and sketches onto separate canvases and embed these canvas content into an SVG. This transformation process includes handling CORS and tainting issues, adding grids to locate annotations, and turning the code editor to grayscale to highlight the sketches. The system then considers the annotations alongside previous iterations of sketch editing and the relevant codebase as part of the input context for code generation. After the code is generated, a difference algorithm is employed to only update the changed sections of the code [6]. The user can press a "Run" button to execute the code, with text or image results shown in the console panel underneath. Users can annotate any executed results as part of their sketches.

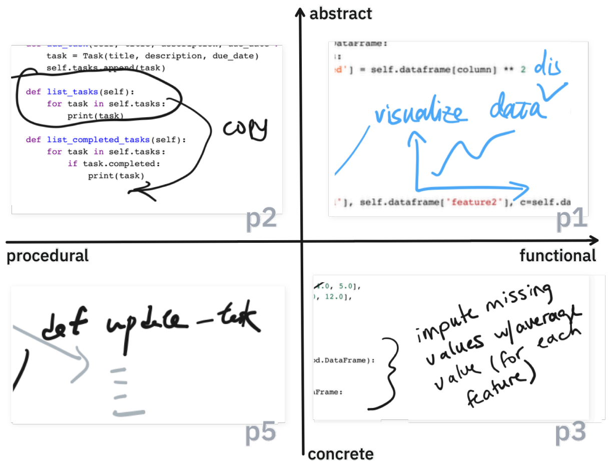
3 EXPLORATORY STUDY

We recruited six participants (1 left-handed), aged 23 to 28, with 4 identifying as women and 2 as men. Participants were recruited through convenience sampling and received $30 for completing the study. All participants had 2-8 years of programming experience and had used ChatGPT or Copilot 3-12 times per week. We designed three coding scenarios, each with two tasks that required specific edits to reach the goal for each scenario. These scenarios covered basic Python programming with object-oriented programming, machine learning with functional programming, and data engineering with declarative programming. A starter code was provided for each task, requiring edits in more than two areas. For example, a task was to extend a class to handle data points with categorical features, requiring changes to current methods to encode features and modify distance calculations accordingly. All tasks were pre-tested to ensure that GPT-4o could not generate the correct code immediately. Participants were assigned 2 out of 3 scenarios that they were more familiar with based on their specifications in the screening questionnaire to avoid relying solely on natural language prompts. They completed 2 SCENARIOS × 2 TASKS each in 50 minutes. Afterward, there was a post-study survey, including UMUX-LITE, NASA-TLX, and a 7-point Likert scale questionnaire evaluating factors such as the clarity of mapping between sketches and edited code and the ease of iterating on sketches. Finally, a semi-structured interview gathered qualitative data on their general experience, challenges encountered, and suggestions for system improvement.

4 RESULTS

All but two participants completed the four assigned tasks, the two participants failed to complete SCENARIO2-TASK2 within the assigned time. The clarity of the effect of their sketches on generated code (Mdn = 3.5, SD = 1.83) and the ease of iterating on sketches (Mdn = 4, SD = 2.34) was rated low. This aligns with our interview results, where participants noted unclear mapping between sketches and edited code, especially with multiple annotations. Four participants expressed a desire to see their annotations stick to the corresponding changed code segments.

Personalized Workflow. We observed participants gradually develop a personalized workflow for editing code with sketches. P2 found that breaking down tasks into very low-level details was



```
    task = Task(title, description, due_date)
    self.tasks.append(task)

def list_tasks(self):
    for task in self.tasks:
        print(task)

def list_completed_tasks(self):
    for task in self.tasks:
        if task.completed:
            print(task)
```

```
ataFrame:
:
d'] = self.dataframe[column] ** 2
'], self.dataframe['feature2'], c=self.da
```

```
.0, 5.0],
, 12.0],

d.DataFrame):

Frame:
```

Figure 2: The classification of sketched annotations from participants situated in a quadrant with two spectrum, ABSTRACT-CONCRETE and PROCEDURAL-FUNCTIONAL.

ineffective for AI interpretation, while P5 emphasized the need for smaller task pieces for better system understanding. Participants sometimes wrote higher-level instructions first when unsure about the solution but had a rough idea of where the code edits should happen and what the *"shape of the code looks like"* [P4]. After evaluating the generated code, they then added annotations for lower-level code editing based on their approaches in mind.

Types of Sketches. Participants used similar sketches for different purposes, such as arrows pointing to context [P1] or targets of changes [P4] (Figure 1b-e). Overall, the sketches could be situated in a quadrant with two spectrums (Figure 2): ABSTRACT-CONCRETE and PROCEDURAL-FUNCTIONAL. The ABSTRACT-CONCRETE spectrum describes whether the annotations are abstract symbols or graphs versus concrete written text. The PROCEDURAL-FUNCTIONAL spectrum classifies the target of the annotations, ranging from procedural steps describing how the program should be structured and run to functional descriptions specifying how the program should work. Participants often combined these aspects, drawing graphs and adding arrows to refer to certain data attributes, specifying both functional and procedural terms.

Sketch as a Tool. Additionally, we observed that participants considered sketches as functional "tools" that could be reused [8], not just as static digital ink drawings. All participants expressed that they could use different sketches to achieve the same effect, choosing which sketch to use based on the environment, such as available white spaces. They also reused their sketches to convey the same effect; for instance, an arrow used to add a function to a target code section was reused by P3 to add another function.

Future Work. This poster takes the first step towards understanding the idea of *code shaping* through a prototype developed in an exploratory user study. We plan to iterate on the design based on the insights gained to support programmers in the iterative process of translating thought into sketches to code editing.

REFERENCES

[1] Christine Alvarado and Randall Davis. 2007. Resolving ambiguities to create a natural computer-based sketching environment. In *ACM SIGGRAPH 2007 Courses* (San Diego, California) *(SIGGRAPH '07)*. Association for Computing Machinery, New York, NY, USA, 16–es. https://doi.org/10.1145/1281500.1281527

[2] Xiaofan Chen and Beryl Plimmer. 2007. CodeAnnotator: digital ink annotation within Eclipse. In *Proceedings of the 19th Australasian Conference on Computer-Human Interaction: Entertaining User Interfaces* (Adelaide, Australia) *(OZCHI '07)*. Association for Computing Machinery, New York, NY, USA, 211–214. https://doi.org/10.1145/1324892.1324935

[3] Randall Davis. 2007. Magic Paper: Sketch-Understanding Research. *Computer* 40, 9 (2007), 34–41. https://doi.org/10.1109/MC.2007.324

[4] James A Landay and Brad A Myers. 1995. Interactive sketching for the early stages of user interface design. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 43–50.

[5] Leonhard Lichtschlag, Lukas Spychalski, and Jan Bochers. 2014. CodeGraffiti: Using hand-drawn sketches connected to code bases in navigation tasks. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 65–68.

[6] Eugene W Myers. 1986. An O (ND) difference algorithm and its variations. *Algorithmica* 1, 1 (1986), 251–266.

[7] B. Plimmer, J. Grundy, J. Hosking, and R. Priest. 2006. Inking in the IDE: Experiences with Pen-based Design and Annotatio. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. 111–115. https://doi.org/10.1109/VLHCC.2006.28

[8] Miguel A Renom, Baptiste Caramiaux, and Michel Beaudouin-Lafon. 2022. Exploring technical reasoning in digital tool use. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–17.

[9] Sigurdur Gauti Samuelsson and Matthias Book. 2020. Eliciting Sketched Expressions of Command Intentions in an IDE. *Proceedings of the ACM on Human-Computer Interaction* 4, ISS (2020), 1–25.

[10] Sigurdur Gauti Samuelsson and Matthias Book. 2023. Towards a Visual Language for Sketched Expression of Software IDE Commands. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 115–123.

[11] Craig J Sutherland, Andrew Luxton-Reilly, and Beryl Plimmer. 2015. An observational study of how experienced programmers annotate program code. In *Human-Computer Interaction–INTERACT 2015: 15th IFIP TC 13 International Conference, Bamberg, Germany, September 14-18, 2015, Proceedings, Part II 15*. Springer, 177–194.

[12] Zhongwei Teng, Quchen Fu, Jules White, and Douglas C. Schmidt. 2021. Sketch2Vis: Generating Data Visualizations from Hand-drawn Sketches with Deep Learning. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 853–858. https://doi.org/10.1109/ICMLA52953.2021.00141