Taking ASCII Drawings Seriously: How Programmers Diagram Code

Devamardeep Hayatpur
University of California, San Diego
La Jolla, California, USA
dshayatpur@ucsd.edu

Brian Hempel
University of California, San Diego
La Jolla, California, USA
bhempel@ucsd.edu

Kathy Chen
University of California, San Diego
La Jolla, California, USA
ktchen@ucsd.edu

William Duan
University of California, San Diego
La Jolla, California, USA
widuan@ucsd.edu

Philip J. Guo
University of California, San Diego
La Jolla, California, USA
pg@ucsd.edu

Haijun Xia
University of California, San Diego
La Jolla, California, USA
haijunxia@ucsd.edu

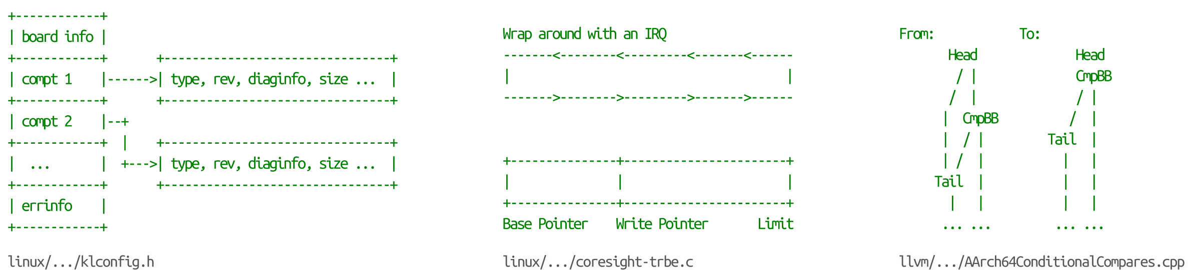

Figure 1: Examples of ASCII diagrams from the Linux Kernel and LLVM codebases [1, 3, 6].

ABSTRACT

Documentation in codebases facilitates knowledge transfer. But tools for programming are largely text-based, and so developers resort to creating ASCII diagrams—graphical artifacts approximated with text—to show visual ideas within their code. Despite real-world use, little is known about these diagrams. We interviewed nine authors of ASCII diagrams, learning why they use ASCII and what roles the diagrams play. We also compile and analyze a corpus of 507 ASCII diagrams from four open source projects, deriving a design space with seven dimensions that classify what these diagrams show, how they show it, and ways they connect to code. These investigations reveal that ASCII diagrams are professional artifacts used across many steps in the development lifecycle, diverse in role and content, and used because they visualize ideas within the variety of programming tools in use. Our findings highlight the importance of visualization within code and lay a foundation for future programming tools that tightly couple text and graphics.

CCS CONCEPTS

• **Human-centered computing** → **Empirical studies in visualization**; *Graphical user interfaces.*

This work is licensed under a Creative Commons Attribution International 4.0 License.

CHI '24, May 11–16, 2024, Honolulu, HI, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0330-0/24/05
https://doi.org/10.1145/3613904.3642683

KEYWORDS

programming, graphical representations

ACM Reference Format:
Devamardeep Hayatpur, Brian Hempel, Kathy Chen, William Duan, Philip J. Guo, and Haijun Xia. 2024. Taking ASCII Drawings Seriously: How Programmers Diagram Code. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24), May 11–16, 2024, Honolulu, HI, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3613904.3642683

1 INTRODUCTION

Computer code is a form of human communication, and most code today is written as text. However, text is a limited media, especially when trying to describe inherently visual ideas. Programmers thus resort to visuals to articulate design decisions to their colleagues, *e.g.* by drawing a diagram on a whiteboard to illustrate relationships and procedures [45]. But these visual artifacts are ephemeral—they are rarely archived or documented digitally [21]. This fragmentation implies that text-based computer code can effectively convey only a portion of its author's thought processes that went into writing that code, thus placing burden on the reader to reconstruct its underlying ideas.

We envision a future of programming where graphics sits alongside text and tightly integrate with it—where developers can use the most fitting representation for any given task. As a first step towards this vision, we wish to understand how text and graphics currently integrate within source code.

To date, the primary media of code is *monospace text*, traditionally nicknamed "ASCII" after the influential and widespread "American Standard Code for Information Interchange" text encoding

format [16, 47], of which more flexible modern encodings such as UTF-8 are a superset [24]. Historically, typewriters and teletypes[1] were *monospace* for mechanical simplicity: each character used the exact same amount of space on the page [30]. The monospace tradition continues in modern text editors for programming.

Given that programming continues to be a text-based activity [14], one may expect there to be few, if any, graphical artifacts within real-world codebases. But instead, the simple grid-like arrangement of monospace text allows for robust alignment between lines of text. Thus we find frequent anecdotes of developers who *appropriate* text through creative uses of monospace text characters to approximate line drawings, which are commonly referred to as ASCII diagrams [54], such as the examples shown in Figure 1.

However, beyond anecdotes, little is known about these diagrams in code. We can only speculate about why and how they are created, what roles they serve, and what information they display. In this work, we seek to reveal the world of ASCII diagrams. We base our inquiry around three research questions:

- RQ1 ***Characteristics.*** What are the key characteristics of the media of ASCII diagrams?
- RQ2 ***Roles.*** How are ASCII diagrams used in the software development workflow?
- RQ3 ***Content.*** What do ASCII diagrams show and how do they show it?

We approach these RQs through qualitative interviews with nine programmers who authored these diagrams, followed by a content analysis of ASCII diagrams gathered from four large open source repositories: Linux, Chromium, LLVM, and Tensorflow. Through the content analysis, we create a design space that summarizes what the diagrams show and how they show it. *To our knowledge, this is the first systematic empirical analysis of ASCII diagrams in code.* Specifically, we contribute:

1. A characterization of the media of ASCII diagrams: what are the features of monospace text and why it is used despite its limitations.
2. An account of the roles that ASCII diagrams play in the software development lifecycle.
3. A design space that reveals the diversity of ASCII diagrams found across four large open source projects.

Our work provides a deeper understanding of visual representations in programming and their role in facilitating communication and knowledge transfer. We hope this study lays a foundation not only for designing future tools for code *documentation*, but also for tools that use visualizations as interfaces for program *construction*.

2 RELATED WORK

While text remains the dominant medium for programming, significant research has investigated the use of visuals for assisting, explaining, teaching, and even doing programming. These investigations are rooted in the thesis that text and visuals, as two different media, have complimentary representational capabilities which can be integrated simultaneously to amplify problem-solving, learning, and communication [22, 46, 60]. Below, we review prior work that uses visual artifacts to assist in problem-solving for software development, as communication and educational tools to teach programming, and as part or as the whole of a programming language.

[1]Of teletype machines, Kjell [38] remarks: *"The better ones smelled of fresh machine oil and chattered pleasantly as they worked."*

2.1 Visuals as Reasoning Tools for Programming and Software Development

Developers use diagrams to make sense of and communicate in software-related tasks [18, 21, 26, 45, 67]. In an interview-based study, Cherubini et al. [21] found that diagrams serve various purposes, including to understand, design and refractor. Yatani et al. [67] reaffirmed many of Cherubini et al. [21]'s findings for open source developers, however, participants did not use diagrams to understand existing code, and rarely used diagrams for design review, which may be because *"design review usually involves updating diagrams, which most of our participants tried to avoid"* [67]. We also know that most diagrams are ad-hoc, improvised, and ephemeral: they rarely find their way into documentation [18, 21]. While interviews with software developers have shown the multifaceted roles diagrams play in software development workflow, a systematic understanding of their representation is impractical from interviews alone, given that many sketches are unintelligible to an outsider and they rely on the author's memory, speech, and contextual cues [27]. Meanwhile ASCII diagrams, being permanent artifacts in large collaborative codebases, should have enough contextual information in the surrounding code file for a viewer to understand them.

Prior work has also studied design activities as they unfold. Mangano et al. [45] found a diversity of visual artifacts created during collaboration on software tasks. These included lists, tables, GUIs, entity-relationship (ER) diagrams, class diagrams, code, drawings, and domain-specific sketches [45]. Dekel and Herbsleb [27] studied 3-6 hours of design activity, and discovered how developers deviate from UML notation, *e.g.* class diagrams were frequently augmented with control- and data-flow [27]. These observational studies are necessarily small scale and self-contained sessions: Dekel and Herbsleb [27] studied seven pairs, and Mangano et al. [45] studied eight groups. As Dekel and Herbsleb [27] note their goal was not to *"catalogue them [diagrams] or to elicit quantitative information, which would have little use in these restricted and unique small settings"*. ASCII diagrams provide an opportunity for large-scale investigations sourced from a diverse population of authors, *e.g.* the diagrams we sourced were from 965 distinct authors.

2.2 Visuals as Communication and Education Tools for Programming

Software visualization systems [17, 28, 50] seek to aid program understanding by representing either static information about code or dynamic information about runtime in a graphical form. These tools are prevalent in pedagogical contexts, where they may help students develop useful mental models of how programs work [33, 34, 43]. For example, Python Tutor visualizes the step-by-step execution of programs by showing the manipulation of variables in memory to help students learn the execution mechanisms of programming languages [33]. Although some ASCII diagrams may be curated snapshots of software visualizations, *e.g.* examples of runtime values, we find that not all are. Unlike software visualization tools, which only surface specific types of program information

in specific representations determined by the developers of visualization systems, programmers use ASCII diagrams to communicate any information that they consider important and relevant to the readers. As we will show, the concepts depicted in ASCII diagrams vary widely, examples include: description of the targeted problems, specific data structures, the execution logic, and specific test cases. Moreover, ASCII diagrams can document outside information, *e.g.* external specifications, whereas software visualizations can only re-represent information already present in the codebase.

2.3 Visual as Programming Language

Visuals have also been integrated into programming environments. Some environments have replaced portions of code with richer graphics, *e.g.* prettified math notation [40], or even domain-specific GUIs such as color pickers [48] or circuit editors [12, 15]. Other systems visualize runtime objects that may be directly manipulated to edit the program, as in Pygmalion [56] and the many works on "programming by demonstration" that followed [25, 44]. The representations in ASCII diagrams may serve as models for what programmers believe are salient representations so that future "programming by demonstration" systems might adapt these representations into novel *interfaces* for editing code.

In the "literate programming" paradigm [39], a program is considered to be primarily an explanatory essay for the human rather than merely code for a computer. In line with this ideal, some environments allow rich formatting of comments. Computational notebooks systems [41] often allow flexible formatting for exposition. For example Markdown [32] cells in Jupyter [51] notebooks allow programmers to freely incorporate images within rich text, although in practice the visualizations in notebooks are more often the *output* of code rather than diagrams for explaining it. The ASCII diagrams we examined seem more often to be the opposite: an explanation for code rather than merely its output. As another example, although more like a traditional editor than a notebook, the Dr. Racket IDE [29] allows users to paste images anywhere in the text, including in comments. This is an intriguing ability, as it allows for richer and more detailed diagrams than ASCII can provide. Nevertheless, as we discuss later, embedded images in source code may need to be supported by *all* the tooling in a team's programming workflow before it can be viable.

2.4 Software Documentation at Large and ASCII Drawings

Software documentation includes a diverse range of materials, from technical standards and design documents to user manuals, all aimed at supporting effective software development and use [57]. Our focus here is on source code documentation. High-quality comments in source code improve program comprehension, maintenance, and debugging activities [59]. Successful open-source projects are consistently well documented, regardless of team size or project scope [13]. However, comments are typically written in freeform natural language, with no tool support. The onus of authoring and maintaining high quality comments is left to the programmers, who often forget to, or ignore them [53, 59, 66]. To address this, various computational approaches have been proposed, including techniques for detecting inconsistencies between code and comment (*e.g.*, JavaDocs [62]) and techniques for generating comments directly from source code [36, 52, 58]. These range from manually-defined templates [58] to neural machine translations [36]. ASCII diagrams differ from natural language because of their visual nature. By understanding the content and practices surrounding ASCII diagrams, we can inform the development of similar computational approaches for diagramming of code.

The study of ASCII drawings themselves is niche. The closest work we find is Yatani et al. [67]'s study, as mentioned earlier, in which they briefly noted that open-source developers turn to ASCII art because it *"can be edited by anyone and handled by current version control systems."* Similarly, Isaacs and Gamblin [37] exploit ASCII's in situ nature to *"meet command line users where they are"* by rendering compact, interactive, package dependency visualizations in ASCII. Our work aims to significantly expand on these passing observations via a systematic investigation into ASCII drawing's media, roles, and content. Other works include Twidale and Nichols [64] analysis of bug reports of user interface (UI) development from several open-source projects, in which they found that ASCII art was included to show elements of the UI alongside text. Building on the idea of utilizing ASCII drawings for UI sketches, Simpson and Terry [55] embed tools in Visual Studio for creating and editing ASCII drawings of UI sketches.

3 INTERVIEW STUDY OF OPEN-SOURCE DEVELOPERS WHO CREATED ASCII DIAGRAMS

We first seek to find the key media characteristics (RQ1) and roles (RQ2) of ASCII diagrams in software development. To do so, we collect a corpus of ASCII diagrams in open-source codebases and interviewed nine of their creators.

3.1 Methods

3.1.1 Data Collection. To collect ASCII diagrams, we performed an initial search within each codebase[2] for single-line and multi-line comments containing at least 20% whitespace characters. This heuristic was based on an observation that ASCII drawings often utilize whitespace for spatial alignment. Our search yielded a total of 141,099 comments. We concatenated these comments into four long files, one for each codebase. The first author, leveraging their initial enthusiasm for this project, scrolled through the entirety of the four files and manually extracted the comments that resembled ASCII drawings, resulting in a selection of 2,162 ASCII diagrams.

What defines an ASCII diagram? A precise definition of an ASCII diagram is difficult. In general, we looked for deliberate use of 2D space. We extracted a diagram when normal text characters were *aligned* to represent a structure (*e.g.* Figure 1). However, we excluded artifacts that were auto-generated (*e.g.* of function arguments), and code snippets that solely differed in indentation or alignment. Our final sample consisted of 2,156 unique ASCII diagrams. Our process is not exhaustive of *all* ASCII diagrams due to the initial 20% whitespace filter and human error, but it provides a sizable sample for study. The sample can tell us about the kinds of

[2]The four code-bases (Linux, Chromium, LLVM, Tensorflow) were chosen for being large, open-source, projects, and having a sizable number of diagrams to study. See Appendix B for code-bases that were searched but not included.

Table 1: Interview participants. Participants were from seven different countries and from three different codebases. Professional programming experience is stated in years. The average years of experience programming in a professional setting was 10.4 years, and the average number of years contributing to open source codebases was 6.7 years.

ID	Repository	Profession	Country	Gender	Professional Programming Experience
P1	`linux`	Principal Software Architect	Israel	Male	20
P2	`llvm`	Principal Software Engineer	US	Male	15
P3	`llvm`	Software Engineer	US	Male	3
P4	`linux`	Software Engineer	Poland	Female	4
P5	`llvm`	Research Assistant	Switzerland	Male	1
P6	`tensorflow`	Senior Software Engineer	UK	Female	8
P7	`linux`	Principal Software Engineer	Canada	Male	11
P8	`linux`	Consulting Member of the Technical Staff	Canada	Male	21
P9	`linux`	Principal Software Engineer	Germany	Male	15

things that *exist* in the wild, but we cannot say the *frequency* that different kinds of diagrams might have.

3.1.2 Participants. We contacted 58 of the authors of the ASCII diagrams, and 9 agreed to be interviewed (see Table 1). We contacted authors who committed their diagrams to the codebase after 2022 to promote recall since our goal was to gain specific insights into their workflow and the creation process. We also filtered out mundane diagrams like simple tables or lists. The time and author of each diagram was sourced from the associated Git commit log message.

3.1.3 Study Procedure. The first author conducted all interviews. Each interview lasted approximately 30 minutes using videoconferencing applications (Zoom and Google Meet). Interviews were semi-structured. Participation was voluntary, and compensation was not promised or provided. Our semi-structured protocol was as follows (see Appendix A for the exact questions asked):

- *(5 min) Introduction.* The interviewer introduced themselves and obtained informed consent to record and transcribe the interview.
- *(10-15 min) Recall.* Participants were asked to read through their ASCII diagram and its surrounding code to re-contextualize themselves to their motivation and process at the time of making the diagram. We asked participants to provide a brief explanation of the diagram, their workflow in creating it, and the diagram's intended audience.
- *(5-10 min) Beyond.* Participants were asked about diagrams other than the one we interviewed them on that they also authored, if the diagram they made were typical or unusual, and on their experience in consuming these diagrams in codebases rather than authoring them.

3.1.4 Analysis. The first author transcribed and coded the interview results. They then synthesized the results into themes. The second and last author reviewed and provided feedback on the codes, the transcript associated with it, and iterated on the themes to synthesize the presentation below.

3.1.5 Reporting. Identifiable information in excerpts from participant transcripts has been removed. We have received permission from the participants whose diagrams we show (as the diagrams are potentially de-anonymizing). The following sections are the themes derived from our analysis. In the interest of clarity and concision, the quotes in the paper are lightly edited to remove filler words.

3.2 Results Overview

Our interviews revealed the key media characteristics (RQ1) and roles (RQ2) of ASCII diagrams in software development workflows. We synthesized the themes that emerged as follows:

1. *Key Characteristics: The Duality of Text and Visual (RQ1).* Perhaps unsurprisingly, but importantly, the key characteristics of ASCII diagrams are their dual nature as text and as visuals. *As text*, the diagrams live in situ within the code and tools already in use. *As visuals*, they illustrate particular concepts more clearly than code or natural language.
   (a) *Text: Living Naturally in the Media of Code.* ASCII diagrams *appropriate* existing textual media by living in situ in the source code within all the tooling already in use when developing software. Thus, there is low friction to creating ASCII diagrams, but, because they are an appropriation of text, participants reported they are tedious to edit (*e.g.* aligning edges).
   (b) *Visual: Adept for Explanations and Overviews.* Participants explained that visual diagrams were often a *better representation* of their mental model of the problem compared to code. Although a diagram is necessarily *less detailed* than code, this allows a diagram to serve as a sort of *thumbnail for the code* to let the reader quickly get their bearings.
2. *Roles in the Software Development Lifecycle (RQ2).* Participants reported using the diagrams for four roles within the software development lifecycle:
   (a) to reify offline work, such as previously undocumented behavior or offline sketches,
   (b) for code review and to help their colleagues verify their work,
   (c) to document for others, and
   (d) to help their future self recall context.

We expand on each of these themes below.

3.3 ASCII Diagrams: The Duality of Text and Visual (RQ1).

Our interviews revealed the single most important, and perhaps also the most obvious nature of ASCII diagrams: they are simultaneously text and visual. ASCII diagrams are text and, therefore, can live freely in the entire infrastructure developed to support text-based programming; they are also visuals and, therefore, can be leveraged to assist in explaining and communicating complex visual concepts hidden in the textual code. We elaborate on the diagrams’ dual *textual* and *visual* natures in turn.

3.3.1 ASCII Diagrams are Text: They Live Naturally in the Media of Code. Text is the *“lowest common denominator”* (P1) in programming tools, and as text, ASCII diagrams live in situ within the programming tools already in use. Diagram authoring is thus easy—it uses the tools at hand—but can still be tedious. Programming tools, optimized for code, occasionally mangle the display of ASCII diagrams. Nevertheless, the benefits of the media at hand caused participants to create the diagrams in ASCII instead of *e.g.* leaving a link in the comments to a richer document.

P1, P2, P5, and P9 all noted the universality of ASCII diagrams textual nature. ASCII diagrams can be rendered in any context featuring a monospace font, including text editors, version control commit messages, bug tracking tools, and terminal interfaces, regardless of the programming language being used, *“the raw file is the view”* (P2). No special tools are needed to make them—these diagrams *appropriate* the tools already in use: *“all you need is a text editor with a monospace font and the spacebar”* (P9). This integration makes them a natural form of visual expression in codebases:

> *“I didn’t even consider [making a rich external graphical image and linking to it] because I felt like ASCII diagrams are just very, very natural...it starts within the comments and with all the tools that you already have with monospace, slashes and stuff.”* (P5)

ASCII diagrams exist in places where images cannot, in situ with the developer’s workflow:

> *“Most of my world is in a text editor through the terminal, so, to have a picture wouldn’t be useful to me because I would then have to have a browser open to point at that or some, you know, heavier program.”* (P8)

The appropriation of text means that ASCII diagrams are simultaneously easy and hard to author. They are *easy to author* in that the same set of text-editing tools that developers use in their everyday work can be used to make these diagrams (P5, P7, P8), one can *“just copy paste it basically...copy paste it down, edit the next line...there’s so many things that are hard to do. I feel like ASCII art isn’t necessarily one of them.”* (P8).

But diagrams are also *hard to author* in that editing them can be tedious (P2, P3, P6, P8). Maintaining spatial alignment is a manual process: *“Especially with aligning all the borders, I think that’s very time-consuming to make sure everything is aligned”* (P6). For this reason, P6, for example, preferred to draw their diagrams without borders (*i.e.* just arrows pointing at text). The brittle monospace structure also makes diagrams a poor fit for parts of the codebase that change often: *“I think is really useful for places where you’re documenting something that’s going to be fairly static. ASCII art is really terrible at documenting something that’s living because ASCII art diagrams are not friendly to update”* (P2).

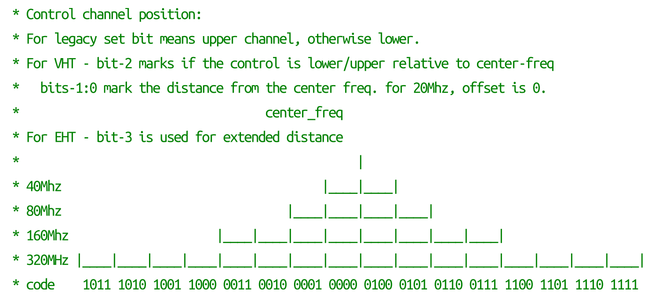

Figure 2: P9’s diagram of Wi-Fi channel band mappings to control channels. It captures the mental reasoning about WiFi channels: *“The channels that we’re using are [...] blocked into like 20 megahertz chunks...mentally you always have this thing [notion], that these are not really like one large channel. They’re like chunked into bit into pieces of 20 megahertz.”* [7]

Although 7 of the 9 participants used only their normal text editor to make their diagram, P2 and P7 leveraged special tools to mitigate the tedium of editing. P2 gladly paid $10 for Monodraw, a diagram editor specifically for ASCII. It allowed them to edit the diagram after-the-fact, provided they had saved the Monodraw file, which proved helpful in a codebase where many contributors were authoring these diagrams: *“the best $10 I could ever spend was having Monodraw...if I want to like enlarge a box, I can enlarge a box, I can move things around...I usually keep the Monodraw files sitting around so that if someone’s gonna come along and ‘Oh I’m gonna have to update this diagram.”’* Meanwhile, P7 used `Org Mode` for Emacs which allows structured editing of ASCII tables.

Ordinary textual tools in software development are optimized for code and may mess up the display of diagrams. Autoformatters may re-indent text (P5) and some tools show commit messages in non-monospace font: *“I’ve never put [an ASCII diagram] in commit messages myself...it’s probably a product of the tools that I use, I don’t think commit messages are an ideal place to put them because a lot of commit messages end up getting rendered in non-monospace fonts”* (P2). Change diffs of wide ASCII diagrams can be difficult to make sense of, *“if the ASCII diagrams are too broad...they’re gonna wrap the lines and then it’s gonna look awful”* (P5).

As an alternative to ASCII, both P1 and P2 mentioned Mermaid [61], a simple textual language for describing and rendering diagrams. The popular code collaboration site GitHub renders Mermaid automatically, so P1 now uses it instead of ASCII art when writing README files. P2’s team also uses Mermaid regularly on GitHub, particularly for charts that are updated often, as updating ASCII is tedious. Nevertheless, P2 remarked that code diffs of Mermaid were hard to understand and P1 still used ASCII in tools that didn’t support Mermaid (*e.g.* terminals and text editors).

To elucidate benefits and drawbacks of the textual medium, we asked the diagram authors why they didn’t instead make their diagram in a richer medium (*e.g.* a Google doc) and leave a URL in the code pointing to the richer visualization. Participants were generally negative about this idea for their diagram in question: external

```
* Topology:
* ---------
*   NS0 namespace          |   NS1 namespace        | NS2 namespace
*                          |                        |
*   +---------------+      |   +---------------+    |
*   |    ipsec0     |----------|    ipsec0     |    |
*   | 192.168.1.100 |      |   | 192.168.1.200 |    |
*   | if_id: bpf    |      |   +---------------+    |
*   +---------------+      |                        |
*            |             |                        |   +---------------+
*            |             |                        |   |    ipsec0     |
*            \------------------------------------------| 192.168.1.200 |
*                          |                        |   +---------------+
*                          |                        |
*                          |                        | (overlay network)
*       ------------------------------------------------------
*                          |                        | (underlay network)
*   +--------------+       |   +--------------+     |
*   |    veth01    |-----------|    veth10    |     |
*   | 172.16.1.100 |       |   | 172.16.1.200 |     |
*   ---------------+       |   +--------------+     |
*                          |                        |
*   +--------------+       |                        |   +--------------+
*   |    veth02    |------------------------------------|    veth20    |
*   | 172.16.2.100 |       |                        |   | 172.16.2.200 |
*   +--------------+       |                        |   +--------------+
```

Figure 3: Some ideas are better illustrated by a visualization than by raw code. P1's diagram of a network scenario would be much harder to comprehend if written only in code [8].

```
 *              +
 * |            +--------+         +--------|
 * |            |  rgn2  |         |  rgn1  |
 * +------------+--------+---------+--------+
 *              ^
 *              |
 *              Aligned address boundary
 *
 * The allocation direction is top-down and region arrays are sorted from lower
 * to higher addresses, so the new region will be the first entry in
 * memory.reserved array. The previously reserved region does not get modified.
 * Region counter and total size get updated.
 */
static int alloc_top_down_disjoint_check(void) { ... }

/*
 * A test that tries to allocate memory when there is enough space at the end
 * of the previously reserved block (i.e. first fit):
 *
 *  |              +--------+-------------|
 *  |              |   r1   |      r2     |
 *  +--------------+--------+-------------+
 *
 * Expect a merge of both regions. Only the region size gets updated.
 */
static int alloc_top_down_before_check(void) { ... }
```

Figure 4: Diagrams are lower detail than code, allowing them to serve as thumbnails to orient the reader. P4 diagrammed test cases so that readers could just glance at the diagrams to understand the memory layout tested in each scenario [9].

diagrams would be not worth the complication (P4, P5) particularly for simpler concepts (P6, P7), harder to share, *e.g.* the need to setup permissions (P3, P4, P6, P9), at risk of going out of sync with the code (P2, P8, P9), harder to author (P8, P9), and incompatible with the tools they already use (P1, P8). These responses suggest they used ASCII for the opposite reasons: in situ text was simpler, easier to share, and less likely to rot compared to external documentation.

> **Takeaway:** ASCII diagrams are *text*: they are naturally viewed, created, and manipulated in situ within one's existing programming workflow and collaboration tools.

3.3.2 ASCII Diagrams are Visual: Adept for Explanations and Overviews. Due to their textual nature, the dual characteristic of ASCII diagrams is that they are *visual*. Compared to natural language or code alone, the visual nature of diagrams makes them *a better vehicle* to convey certain programming ideas, and to summarize code.

P1, P2, P4–9 reported that diagrams were more direct representations for their problem domain than natural language or code: *"It helps visualize what this test is for, because when I try to explain it in words, it was really complicated and I thought that this is an easier way for me to convey what I wanted to do with these tests"* (P4). Diagrams better portray the author's mental model so that readers can see the same view as the author of the code—the diagram should *"record the mental model I used when I created this"* (P1).

For example, Figure 2 shows a diagram of Wi-Fi bands that P9 helped author. It shows a *"critical part of how an expert would have a mental picture...physically it's a little more complex, but that's sort of the mental image you would have"* (P9). P8 and P4 also specifically mentioned how visualizations clarify their problem domain:

> *"Especially in my area, memory management, we want to explain race conditions...columns and spacing is used to show the ordering in which something could go awry, and having that spacing and the columns really helps in the race condition explanation. Often people write it and it is very difficult to follow if done correctly and impossible to understand otherwise."* (P8)

> *"I'm really referring to these blocks because I'm seeing them as blocks and how they lay in memory."* (P4)

Code itself is not a suitable representation for some concepts, for example P6, looking at a program graph as a diagram makes it *"pretty clear to me how the graph looks like. I mean if I look at the code I can figure it out, but it would be much more time consuming."* Similarly, P1 said *"if you look at the code [under the diagram], it looks pretty horrible...it is not very palatable"*, thus they drew out the network topology diagram shown in Figure 3. Different kinds of code may be more suitable for diagrams: *"there's certain areas of the codebase that definitely benefit from having small diagrams embedded in the code. Especially like when we look at things like...optimizations that manipulate the control flow graph or manipulate the SSA instruction graphs and these are all just graphs"* (P2).

Despite its flexibility, ASCII is not a full-featured graphical medium. For example, P5 wished to show overlapping shapes, where *"some color would actually help in understanding what's happening to be able to distinguish the two polytopes"* (P5), but ASCII lacks color.

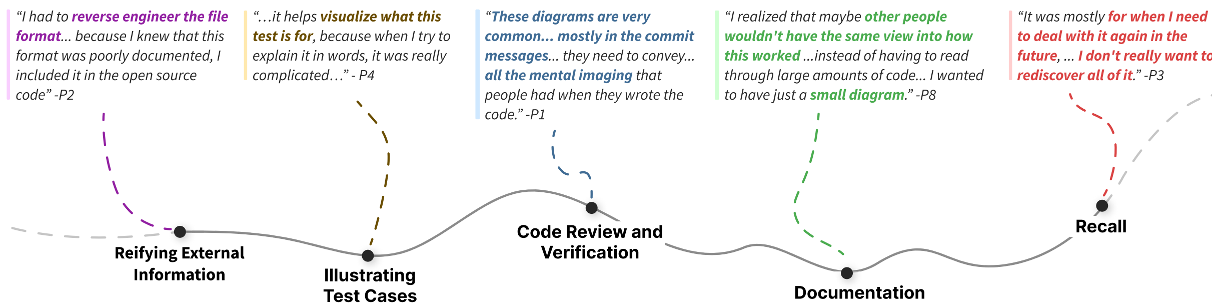

Figure 5: Participants mentioned five roles in the software development lifecycle for which ASCII diagrams are useful: (1) to reify offline information, (2) for illustrating test cases, (3) for code review, and to document both (4) for others and (5) themselves. Shown are representative quotations from participants for each role.

Another theme that emerged was that diagrams do not convey the same level of detail as code, but this also makes them ideal for helping orient the reader with needed higher-level context. P1, P4–9 reported that diagrams are more inviting and accessible than code. They help convey the gist and act as a sort of thumbnail for the code. For example, P4, who documented a series of test cases shown in Figure 4, wanted each test case in the file to be approachable:

> *"This [diagram] is also for understanding the big picture...this is the first impression that allows you to dig deeper...I wanted people just to scroll and see 'Okay. So this is this scenario', or maybe that one. So, instead of reading through a paragraph of text, people can just look and see 'Okay, that's that.' "* (P4)

Similarly, P8 drew their diagram because they did not want future readers to have to scrutinize all the code to understand its functionality. The diagram was small in comparison to the code:

> *"So instead of [other people] having to read through large amounts of code to to try and piece together what's going on, I wanted to have just a small diagram."* (P8)

Diagrams serve as *"simplified versions"* (P4) that help readers orient.

> **Takeaway:** ASCII diagrams are *visual*: visualizations are often a better projection of the developer's mental model than code or natural language. They are necessarily less detailed than code and thus can serve as an approachable entrance to it, as a sort of *thumbnail for the code.*

3.4 Roles of ASCII Diagrams in the Software Development Lifecycle (RQ2)

Participants reported that ASCII diagrams served a variety of roles within the software development lifecycle. Figure 5 provides a birds-eye overview of how they used diagrams (1) to reify offline work, (2) to illustrate test cases, (3) for code review and verification from colleagues, and to document both (4) for others and (5) themselves. We elaborate on each of these in turn.

3.4.1 Role 1: Reifying External Information Gained During Problem Understanding. Participants reported two kinds of scenarios where information in their ASCII diagram originated from outside the code: (a) sketches they created in notebooks or whiteboards to understand problems were sometimes reified into ASCII, and (b) diagrams were also used to note undocumented external behaviors the developers had carefully deduced (*e.g.* by reverse-engineering a file format). Interestingly, the participants did *not* mention using ASCII as a medium for *initially working out* ideas. We speculate that whiteboards or notebooks are better than ASCII for exploratory sketching, perhaps because *e.g.* aligning ASCII text can be tedious. But because of our small sample size we cannot definitively rule out the possibility of ASCII being used for working out ideas in earlier design phases. We discuss the two kinds of scenarios reported—reifying offline sketches and undocumented information—below.

P3, P4, P5, and P8 each reported diagramming in an external medium, such as a whiteboard or a notebook, and then translating into an ASCII diagram. P3 experimentally worked out the bit-by-bit behavior of a memory region, writing down their findings in a notebook. Later, *"almost certainly...after I wrote the code"* they translated their findings into the ASCII diagram shown in Figure 6. P8 additionally noted that they collaborated with their colleague on a whiteboard to develop a data structure that was later written in code and diagrammed in ASCII. Here, the developers worked on an offline drawing medium first before creating the ASCII diagram.

An unexpected use of ASCII diagrams reported by P2 and P3 was to document some undocumented *external* information. More than once, P2 found they needed to reverse engineer the layouts of undocumented file formats. Similarly, P3 discovered undocumented differences in floating point exception handling behavior between Windows and Linux, which they manually tested to characterize the details. The results of P2's and P3's experiments were documented in ASCII. Figure 7 shows one file format P2 reverse engineered, and the previously mentioned Figure 6 is the ASCII documentation of the bit-level configuration deduced by P3.

3.4.2 Role 2: Illustrating Test Cases. Both P1's and P4's diagrams (Figure 3 and Figure 4) were attached to test cases. The diagram served to describe the setup under test. P1 made the diagram because the setup *"isn't trivial to understand from the code."* And, P4 wanted readers to quickly see what was tested in each case: *"I really wanted people just to scroll and see 'Ah, okay, so this is this scenario, or maybe that one!' "*. Diagramming of tests is not uncommon: in our content analysis described later, 11% of the diagrams we examined had to do with a test case.

```
The MXCSR format is the same information, just organized differently.
the fenv_t struct for windows doesn't include the mxcsr bits, they mu
generated from the control word bits.

  Exception Masks---+           +---Exception Flags
                    |           |
 Flush-to-zero---+  +----+ +----+
                 |  |    | |    |
                 FRRMMMMMMDEEEEEE
                  ||      |
                  ++      +---Denormals-are-zero
                  |
                  +---Rounding Mode
```

Figure 6: P3's diagram of a region of memory that stores information about floating point exception handling in operating systems. This information was originally worked out on paper and reverse-engineered via trial-and-error: *"When testing, I did not find any like documentation on it. I just went through it...I actually tried setting different bits, and running it and seeing what changed"* (P3) [10].

```
// The DXContainer file format is arranged as a
// parts are similar to sections in other obje
// structure is roughly:

// ┌──────────────────────────────┐
// │            Header            │
// ├──────────────────────────────┤
// │             Part             │
// ├──────────────────────────────┤
// │             Part             │
// ├──────────────────────────────┤
// │             ...              │
// └──────────────────────────────┘
```

Figure 7: P2's diagram of a previously undocumented file format they reverse engineered: *"I'll have one window open of a hex dump of the file and be looking at the hex codes of what's where and trying to draw out boxes...it really helps me to be able to visually construct how these things go sequentially"* (P2) [11].

3.4.3 Role 3: Code Review and Verification from Colleagues. P1, P2, P4, P7, and P8 reported the importance of ASCII diagrams for code reviews and eliciting feedback. For example, after understanding the problem and formulating an implementation, P2 and P4 each used their diagram to communicate their understanding of the problem for code reviewers to be able to understand and verify:

> *"Someone had to review that code change and I spent three weeks or something reverse engineering this complicated binary encoding [...] Well now I need to explain it well enough to someone else that they can look over my code and verify that my code does the right thing and that my understanding of the problem matches reality."* (P2)

P1 noted that ASCII diagrams, at least in the networking sub-area of the Linux kernel, are commonplace in commit messages to communicate to reviewers the reasoning behind changes.[3] ASCII diagrams can also be used to elicit feedback in asynchronous emails, like in an RFC,[4] rather than presenting something as just uninviting code:

> *"Like if this [code associated with P8's diagram] went out as an RFC, and I was really expecting people to be like: 'I don't know...I'm not reading...like 10,000 lines of code to see if your idea is worth doing.' Whereas if you say, 'Hey, here's 5 lines of a picture,' they're like, 'Oh, a picture, okay!' "* (P8)

3.4.4 Role 4: Documentation for Others. P1, P2, P4, P7–P9 reported authoring their diagram to document the technical ideas for colleagues and other readers of the code. For example, P2 worked on a proprietary compiler project in which many collaborators were using ASCII to document graph-like data structures:

> *"[we] did a whole lot of graph transformation algorithms and we were very aggressive about documenting the program state graphs in our code and at each transformation step. That was something that was really useful for our team to be able to to communicate and understand."* (P2)

As another example, P4 chose to include variable names in their diagram so *"someone who's looking at the diagram can later use it while reading the code. So, I'm including R1 or R2 which are the names of the variables."* P8, who created a novel data structure with their colleague, added the ASCII diagram for other readers because *"I realized that maybe other people wouldn't have the same view into how this worked."*

Documentation is only useful if it is correct (*"wrong documentation is worse than no documentation"* - P8). We asked P2 and P6–P8 if the tedium of updating ASCII diagrams might make them more likely to be neglected and become stale. P6 and P7 disagreed that updating ASCII diagrams was difficult and they expected discrepancies would be caught in code review. P2 agreed that updating was difficult, but later explained that in situ diagrams were less at risk of going stale compared to external docs, *"if I change the code and I change the behavior, the documentation is right there and so I can update that documentation."* Similarly, P8 observed that when you review a patch for parts of a codebase you are familiar with then *"you're not looking at the documentation"* so you may forget to check if it is updated. Indeed, P8 recalled an instance as a reviewer where they missed that a patch did not update the documentation. The importance of developer attention to documentation was also highlighted by P9, in response to a different question they noted that *"documentation and comments generally, unless someone's actually paying attention, become stale."* Staleness is certainly a risk for documentation *external* to the code. Both P2 and P8 noted that the external documentation of their projects (LLVM and the linux kernel, respectively) had some information that was old, *"like a*

[3] P1 demonstrated this commonality by scrolling through the most recent commits in the networking sub-area of the Linux kernel and quickly found one: https://github.com/torvalds/linux/commit/f9c4bb0b245cee35ef66f75bf409c9573d934cf9

[4] RFC stands for "Request for Comments," a kind of technical document for proposing standards and discussing development and operation.

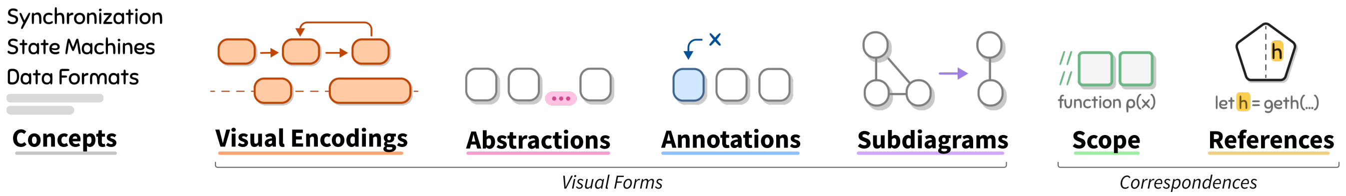

Figure 8: An iconographic overview of our design space's seven dimensions.

decade out of date" (P2). P2 opined that this may in part be because developers care more about code than prose:

> *"If you put documentation apart from the code, one of the problems that you get is that they're two living independent things...the people that become software engineers and become successful as software engineers like to write code. They don't want to write documentation...if you have documentation over there and code over here, they're going to change the code. They're not going to remember that the docs exist."* (P2)

As mentioned previously, in situ documentation such as ASCII diagrams might be less prone to rot than external documentation (P2). P8 opined on another potential difference with in situ documentation: external documentation should perhaps be targeted at *users* of code, whereas in situ documentation may be more appropriate for the code's *developers*:

> *"I believe in the first patch that perhaps this [diagram] was in the kernel docs, but as the [implementation of this] tree matured, it became apparent that I needed a separation between users of the tree versus developers of the tree for the documentation...One that says this is how you use it, this is how it's supposed to be used, and another set that says this is how it works."* (P8)

3.4.5 Role 5: Self-Recall. ASCII diagrams were not just for authors to convey an idea to *other* people. P3, P6, P7, P8, and P9 also specifically mentioned their diagram was useful for *themselves* to regain context when they revisit the code months or years later:

> *"Even if I wrote the code, after like two months I probably won't remember how the graph looks like."* (P6)

> *"[I made the diagram] mostly for when I need to deal with it again in the future, because I know that I'm not going to be able to remember all of it and I don't really want to rediscover all of it...I'm not gonna remember in like a year how all this stuff is organized."* (P3)

These participants created diagrams to help themselves *remember.*

Takeaway: Developers create ASCII diagrams for several roles in the software development life cycle: (1) to bring in outside knowledge (either from offline sketches or to reify undocumented behavior), (2) to illustrate test cases, (3) for code review, (4) for documentation for others, and (5) to help themselves remember.

4 DESIGN SPACE OF ASCII DIAGRAMS

Having understood the characteristics of ASCII diagrams as a medium—the duality of text and visual for tool agnostic diagramming—and the roles that ASCII diagrams play in the software development cycle—for documentation and explanation for various stakeholders—we now seek to understand the design of diagrams in this unique medium. We opt for a thematic analysis [20] to map the design space of these diagrams, using a sample of the aforementioned 2,156 ASCII diagrams for our analysis. We aim to answer the following questions with our analysis:

(1) What concepts are illustrated?
(2) What visual forms are used?
(3) How do diagrams correspond to the surrounding code?

The output of our analysis is a *design space* for classifying ASCII diagrams' concepts, visual forms, and correspondences to code. Our dataset of coded diagrams is available publicly at https://asciidiagrams.github.io/.

Table 2: Repository statistics. The Linux kernel contained the largest number as well as the largest ratio of diagrams to lines of code. Note: the LOC includes blank lines.

Repo	LOC	Diagrams	LOC/Diagram
Chromium	56,697,921	428	132,471
Linux	35,415,763	1,386	25,552
LLVM	28,275,527	220	128,525
Tensorflow	6,618,475	122	54,249

4.1 Methods

Four members of the research team (hereafter referred to as "coders") collaborated on the analysis. All coders had backgrounds in computer science.

4.1.1 Theoretical Sampling. To establish a smaller sample from which to develop a codebook, the 2,156 diagrams (see Table 2) were narrowed down as follows. The first author categorized each diagram into one of 15 visual types, such as memory layout, table, plot, state machine, and "unknown" for unique diagrams. These tentative categories were developed inductively and refined through multiple reviews of the dataset. We then randomly sampled up to 20 diagrams from each category, along with all 334 unknown diagrams. This resulted in a smaller sample of 507 ASCII diagrams that was less skewed towards any particular visual form.

Concepts

Resource Management
Memory

Data
Data Structure
Data Format
Bit Interpretation
Memory Layout

Hardware

Geometry/ Graphics
Interface Sketch

Algorithms/ Data Processing
Math Formulas

Test Case

Synchronization
Threads
Hardware Signal Timing
Queuing/Scheduling

Layout/ Architecture
Actor Interactions
Class Diagrams

Information Flow/ Instructions
Programs
Conditional Control Flow
State Machine
Data Flow

Visual Encodings

Connection
Nodes that are related to each other by a visible connection.
Linear
Tree
Graph
Directed
Undirected

Geometry
Idealized spatial relationships, *e.g.* UI sketches, shapes, etc.

Table
Rows and columns representing an item and an attribute.

Nested
Show visible containment.
Parent

Sequence
Show linear order, including continuous.
Single
Aligned

Math Notation
Mimic formulas with spacing and special characters.
$x_i = \sum_i^n y_i$

Code Annotation
Visualization itself points to parts of source code to elaborate.
A
B
x = [1, 6, 9]

Pictorial
Pictures showing appearance of 3D objects recast in ASCII, e.g. isometric projections, faces, etc.
On

Annotations

Point
Selects an object with a visible annotation (e.g. an arrow).
target

Multi-Point
Selects multiple objects with lines stemming from a one label.
target

Range
Selects a range or an interval.
start

Legend
Elaborates on the meaning of symbols in diagram.
* marked

Abstractions

Unpatterned Elision
No inference can be made of the elision.
Fragment

Patterned Elision
Pattern implied from a few instances.
Sequential

Subdiagrams

Multiple Representations
Same concept with different visual encodings.
x y

Multiple Scenarios
Different scenarios with same visual encoding.
a
b
Over Time

Scope

File
Class
Multiple Functions
Function
Multiple Statements
Statement

References

Identifiers
Names shared in the diagram and the code, variable names, function names, class names, etc.

Constants
Exact values (e.g. `0xfff`) shared between the diagram and the code.

Expressions
Code expressions in diagram (e.g. `x-5`) that are also used in the actual source code.

Figure 9: Our design space for ASCII diagrams, listing the possible codings for each of the seven dimensions.

4.1.2 Developing a Codebook. We employed thematic analysis to analyze the diagrams, inductively grouping them into coherent categories along the dimensions outlined: *concepts*, *visual forms*, and *correspondences*. To establish a coding guide, we initially set aside 100 diagrams from the 507. Over a six-week period, each coder independently analyzed subsets of 20 diagrams. We held regular

meetings to share interpretations and discuss emerging themes. The first author consolidated these insights into a coding guide, resolving any conflicts through discussion.

Given specificity of each diagram to a particular problem and domain, the *concept* dimension was handled specially. The four coders examined the remaining 407 diagrams (~100 per coder) to discern the specific concepts illustrated (*e.g.*, abstract syntax tree, IPC connections, graph partitioning). The coders then independently clustered these concepts and then convened over lengthy discussions to distill them into the overall concept categories.

The dimensions, including specific concepts, were not derived from prior work but established inductively in the data collected.

4.1.3 Coding. The primary contribution of the content analysis is the codebook itself, since it summarizes what *exists* in the ASCII diagrams. We cannot make precise claims about the frequencies. Regardless, for establishing inspection in the online dataset and for exercising the codebook, three coders independently coded different subsets of the diagrams. In total, 504 of the 507 diagrams were coded (three machine-generated diagrams were excluded). Diagrams that any coder was uncertain about were flagged and discussed until consensus was reached. Prior to coding, a sample of 20 diagrams was coded by all three coders to calibrate their interpretations and resolve any potential coding discrepancies. The coding process took approximately 14 hours per coder, with each coder making approximately 10k judgments (168 diagrams per coder $\times$ 59 sub-dimensions).

4.2 Design Space Summary

We converged upon seven top-level dimensions for our design space, summarized iconographically in Figure 8. Together, the seven dimensions describe a diagram's concepts (1 dimension), visual forms (4 dimensions), and correspondences to code (2 dimensions), as explained below. The complete listing of possible codes for each dimension are shown in Figure 9.

*What **concepts** does the diagram convey?*

(1) *Concepts* (25 codes) characterizes the technical ideas that the diagram illustrates, *e.g.* synchronization, an architecture layout, a hardware description, etc.

*What **visual forms** are used to describe the concept?*

(2) *Visual Encodings* (12 codes) describe the basic visual patterns used, *e.g.* a directed graph or a linear sequence.
(3) *Annotations* (4 codes) are labels connected to the main diagram with explicit arrows or symbols.
(4) *Abstractions* (4 codes) characterizes elision of information with *e.g.* ellipses.
(5) *Subdiagrams* (3 codes) indicates the use of multiple subdiagrams to *e.g.* show before and after.

*How does the diagram **correspond** to the surrounding code?*

(6) *Scope* (6 codes) characterizes whether a diagram applies to *e.g.* a whole file or just a single function.
(7) *References* (3 codes) notes whether the diagram refers to any identifiers, constants, or expressions in the code.

The most salient finding highlighted by our design space is that **ASCII diagrams are diverse**, despite the medium itself being primitive. They are diverse in both in the concepts they illustrate and the visual forms they use. Although each concept in Figure 9 is rather generic, we still discovered a large variety among the diagrams. Similarly, the visual forms are also diverse, particularly the possible visual encoding. ASCII diagrams may be made from a simple medium, but they are not just one thing.

> **Takeaway:** ASCII diagrams are not just one thing. They represent *diverse concepts* with *diverse visual forms*. ASCII diagrams are *connected to the code* explicitly by references and implicitly by juxtaposition.

4.3 Illustrative Examples

Let us illustrate how our design space describes three ASCII diagrams. Taken together, these three diagrams meaningfully exercise all seven of our design framework's dimensions. Specific codings for a diagram (drawn from the possible codes in Figure 9) are indicated by a *filled background*.

4.3.1 Frame Graph. Figure 10 is a diagram from the Chromium project depicting a test for whether a traversal through a data structure happens as expected. The structure represents nested web pages and is quite complicated: four separate trees are connected via special edges that form a directed graph *between* the trees. The labels *root1*, *root2*, *root3*, and *root4* are the roots of the four trees, the numbers below are each tree's children, and the remaining lines and arrows are the special edges forming the directed graph. (Semantically, each tree represents the nested `iframe` structure of a web page, while the special edges between trees represent another kind of web page nesting that has stricter communication permissions than `iframes`.[5])

*What **concepts** does the diagram convey?* At the most specific, this diagram depicts an "opener graph" of "FrameTrees," but among the more general *Concepts* of our design space, this diagram depicts a *Data :: Data Structure* and a *Test Case*.

*What **visual forms** are used to describe the concept?* This diagram uses lines to show connection relations: both the *Connection :: Tree* pattern for the four trees and the *Connection :: Graph :: Directed* pattern between the trees—these are the *Visual Encodings*, the basic visual patterns, of this diagram. The remaining three visual dimensions are not represented in this diagram. All the labels in this diagram are in place (*i.e.* lack a connecting arrow) and so are not considered *Annotations*, the diagram does not indicate any *Abstractions* by hiding information with *e.g.* ellipses, and it does not contain multiple *Subdiagrams*.

*How does the diagram **correspond** to the surrounding code?* In this case, the diagram only describes the situation for a single *Function*, that is its only *Scope*. The diagram also uses specific text that *References* the code: the names `root1` etc. are *Identifiers* also used in the code, as are the *Constants* `12`, `13` etc. used to refer to the tree children.

[5] https://chromium.googlesource.com/chromium/src.git/+/refs/heads/main/docs/frame_trees.md

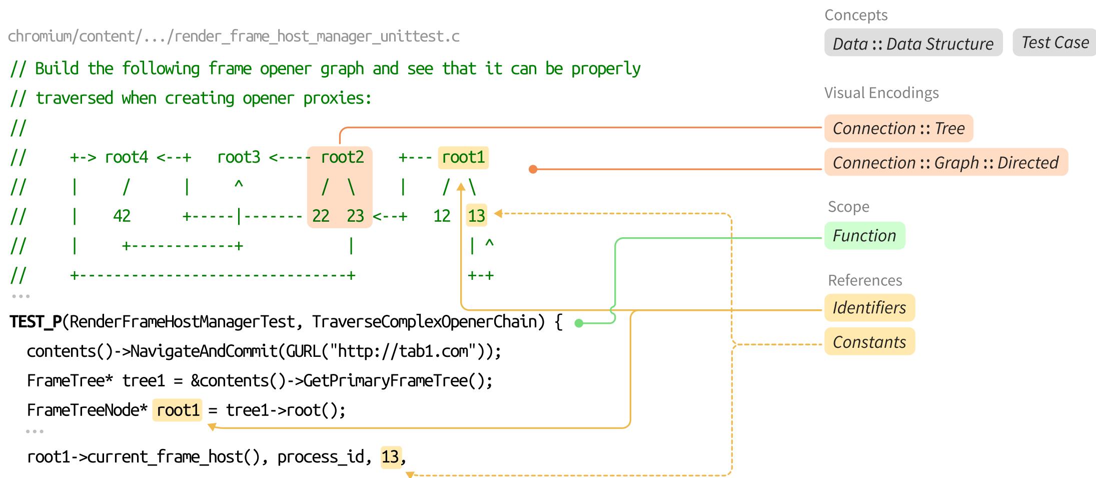

Figure 10: Test case to check if a graph can be correctly traversed in Chromium, with the codes under our design space [4].

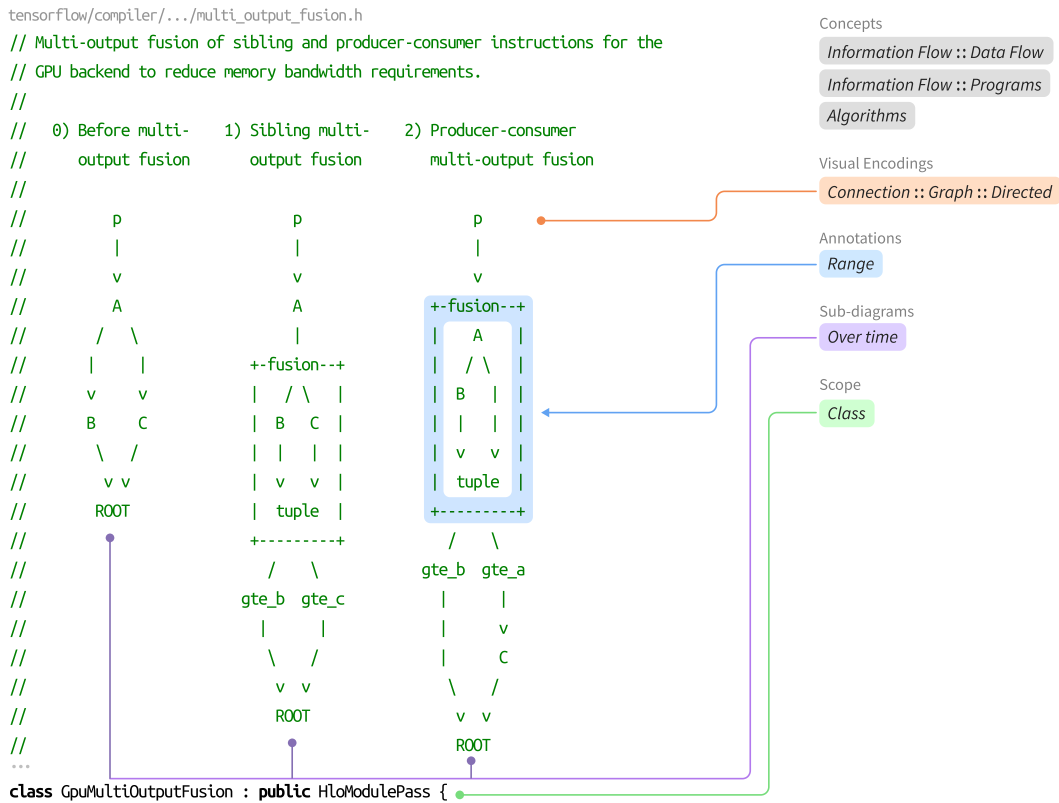

Figure 11: A diagram in the Tensorflow codebase depicting how a computation graph changes when operations are fused together for efficiency on the GPU, with the codes under our design space [5].

4.3.2 Multi-Output Fusion. For our second example, we consider the ASCII diagram from Tensorflow in Figure 11, which depicts how operation fusion changes a computation graph. Unlike the prior example, this diagram contains *Annotations* and *Subdiagrams*. For *Annotations*, the `fusion` label uses a box to select a *Range* of nodes in the graph. This diagram also has three *Subdiagrams*: the first depicts an initial scenario, while the other two depict two versions of fusion applied to that initial scenario. We coded such before and after snapshots as *Multiple Scenarios :: Over Time*. The codings for the remaining dimensions for this diagram are given in Figure 11.

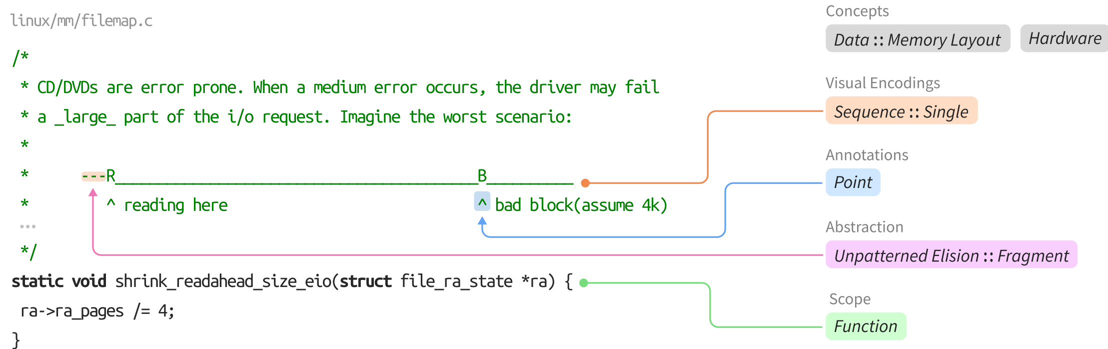

Figure 12: An ASCII diagram from the memory-management sub-part of the Linux kernel handling disk read errors, labeled with the categories from our design space [2].

4.3.3 Disk Read Error Handling. As a final example, the diagram in Figure 12 from the Linux kernel illustrates a potential disk read error when reading from a CD/DVD. It depicts a scenario where there is a read operation at (R) with a bad block (B) within the read-ahead distance after (R), causing the read for (R) to fail. The associated function simply reduces the size of the read-ahead window in hopes that will cause the read to succeed. The full coding of this diagram is given in Figure 12. Most notably, this diagram involves *Abstractions*: the diagram explicitly indicates the presence of elided information. Here, the diagram indicates that the sequence shown is only a *fragment* of the larger memory sequence. This "fragment of a larger thing" scenario, with elision on the edges, is coded as *Unpatterned Elision :: Fragment*.

4.4 Summary and Implications

It should be noted that certain dimensions within our design space overlap with existing categorizations. For example, *Visual Encodings* overlaps with classifications of graphical representations, *e.g.* Engelhardt's different types of graphics like *link diagrams*, *tables*, *text*, *maps* [65]. Recently, BlueFish, a relational grammar of graphics, operationalized Gestalt relations (*e.g. nesting* and *links*) to provide a grammar for describing data-driven graphics [49]. *Multiple Scenarios* closely aligns with Tufte's *small multiples*, where different datasets share the same visual encoding and are positioned adjacent to one another for effective comparison [63]. In a content analysis of mathematical formulas, Head et al. identified various annotation mechanisms, including *extent*, *pointer*, and *border* in augmented math formulas [35].

Our categorization also surfaces the in situ nature of ASCII diagrams. They are diagrams attached to the code both by juxtaposition (*Scope*) and, often, by explicitly sharing the same identifiers, constants, and expressions (*References*). Because they are close to code they can reference it, and because they can reference the code they can better describe it. As such, our design space provides a common ground and suggests new possibilities for tools integrating code and visual elements. For example, the de facto *live programming system* shows output of *Data* (*e.g.* a list as `nums: [1,1,2,4,2]`), but none, to our knowledge, show *Abstractions* of the *Data* displays themselves (*e.g.* displays like `nums: [x,...,y]`, which capture entire classes of output). Similar exercises in extrapolation can be made by mixing static, user-created *Annotations* with dynamic runtime displays of *Data*. Or, linking live programming output to various *Scope* code structures beyond statement-level, such as to function calls, multiple statements, classes, and entire files.

4.5 Limitations and Threats to Validity

The coders were all from a computer science background. Concepts that were in other domains *e.g.* hardware or electrical engineering, may not be as carefully represented by the design space. Moreover, we only sampled four codebases, coding a subset of diagrams that were deliberately selected for their diversity of forms. We therefore cannot make generalizable claims about the frequency of different traits, only that these traits exist.

5 DISCUSSION

Our interviews have revealed the prevalence, benefits, and limitations of ASCII diagrams, while our design space provides a shared ground for tools that couple visuals in software development workflows. Most immediately, these investigations suggest (1) the diagrams have a close but tenuous relationship to their code, (2) their appeal and use is heavily dependent not merely on the author's text editor, but on multiple collaboration tools throughout the software development lifecycle, and (3) design opportunities for moving beyond ASCII while preserving the universality.

5.1 Code and Diagram are (Dis-)Connected

ASCII diagrams have a close but tenuous relationship to code. Part of their utility is that they live next to the code they describe. This lets them specifically reference snippets of the code to better explain its behavior. But this close *connection* with code is also, simultaneously, a risk for *disconnection*. Our interviewees expressed concerns about documentation becoming out of sync with the code—even though ASCII diagrams are perhaps more likely to be updated because they are close to the code, programmers may still forget about them because of hyper-focus on scrutinizing the code's function. IDEs are little help: they are oblivious to ASCII diagrams. A rename refactoring, for example, will ignore comments by default. Providing better tooling for *making* and *maintaining* connections between code and diagram may prove pertinent for using diagrams at scale.

Similar concerns of disconnection apply to code comments in general, as documented in previous research [53, 59, 66]. However, the structured nature of diagrams, compared to natural language comments, may make them more amenable to preserving connections between code and documentation. Beyond the correspondences identified in our design space, one specific situation we encountered in both our interviews and content analysis was using the diagram to illustrate the expected result of a test case. Extrapolating from this, a tool could potentially regenerate live visual documentation based on the outcomes of test cases. Bigelow et al. [19] demonstrate similar bidirectional linkages between user-editing and data-driven chart visualizations.

5.2 Beyond ASCII? Or Will ASCII Live Forever?

As a thought experiment, let's ponder: *could a richer tool replace ASCII for diagramming?* For example, what if the programmer's IDE let them embed high-resolution diagrams in situ with the code, and provided an in-editor interface for editing those diagrams? Would that be enough? Maybe. What our investigation highlights is that ASCII works because it is supported across a *wide* variety of the tooling already in place—how uniform is the tooling across the entire development team? The entire team would need to use our hypothetical IDE. Moreover, all the collaboration tools in between would need to support the rich diagrams as well: the diff viewer, the version control, the bug tracker, and the team chat application. It is hard to imagine that *all* these tools would agree on a standardized format for diagrams in code. Some kind of in-browser cloud-based development environment is most likely to succeed at integrating rich diagrams. Like existing cloud-based software suites for "office" tasks, a cloud-based system offers a unified tool set for the entire team. A promising example is computational notebooks [41], which juxtapose code with rich documentation. The popular Jupyter [51] notebook system is in-browser and is already sometimes offered as a cloud service. But, again, it seems that the stickiness of ASCII diagrams arises from of the diversity of tools programmers use. Existing tools are diverse, but they all support ASCII.

5.3 Design Opportunities

We synthesize these insights into a hypothetical proposal for moving beyond ASCII while maintaining its universal power, without resorting to a shared cloud-based IDE. Our imaginary editor would support bidirectional editing of ordinary ASCII art as rich objects and would infer and leverage the connections between the ASCII diagram and its corresponding code.

Bidirectional editing of diagrams as both rich graphic objects and ASCII diagrams. As noted by the interviews, ASCII is the *"lowest common denominator"* (P1). An improved ASCII diagram editor must respect this: its canonical underlying data format must be ordinary ASCII art so that all other viewers and editors can operate as normal. Our idealized editor would therefore store its diagrams as ordinary ASCII in the code, but would be able to interpret the ASCII as rich shapes and lines. That is, users equipped with the editor would be able to view ad hoc ASCII diagrams as clean vector graphics, as well as edit them as a rich diagram, with the editor bidirectionally saving raw ASCII into the file. Users without the tool can see (and edit) the normal raw ASCII art.

Our design space suggests that our rich editor should support the easy creation and rich editing of common visual structures already in use, such as sequences, graphs, tables, and annotations (Figure 9). This aligns with existing research on bidirectional editing of graphics and code [23], and recent tools like Lorgnette [31], which allows projecting code into more user-friendly representations (*e.g.*, editing an ASCII table as an interactive table).

Reify connections between the diagram and code. ASCII diagrams relate to the code they describe. Our hypothetical editor could leverage this to *e.g.* generate an ASCII graph diagram from the dependency structure of the code. In the other direction, if the editor can infer the relationships between an arbitrary ASCII diagram and the code, that would open up many interface possibilities. The editor might highlight when the structure of the visualization seems to no longer correspond to the code. Or, mousing over the diagram can highlight the corresponding part of the code, while clicking the diagram could take the user to that code. Renaming in the diagram might simultaneously rename in the code. Automatic inference of these correspondences for arbitrary ASCII diagrams is, admittedly, a dicey suggestion. Smart inference can break down [42]. Nevertheless, the recent stunning progress in large language models does paint some hope that smart tools of the future might be more capable than smart tools of the past.

6 CONCLUSION

To learn the key characteristics, roles, and content of ASCII diagrams, we interviewed nine ASCII diagram authors and synthesized a design space of diagrams from four large open-source codebases. ASCII diagrams are dually text and visual. As *text*, they are naturally viewed, created, and manipulated in situ in the programming workflow. As *visuals*, they are often a better explanatory representation than code or natural language. They are less detailed than code and can be an approachable "thumbnail" for code to orient the reader. Developers create ASCII diagrams for several roles in the software development life cycle: to reify outside knowledge, to illustrate test cases, for code review, for documentation for others, and to help themselves remember. The design space we derived from content analysis on ASCII diagrams highlights that they represent *diverse* concepts with *diverse* visual forms. ASCII diagrams are connected to the code, often incorporating code snippets in the diagram.

As mentioned at the beginning of the paper, our longer term research goal is not to supplant ASCII for *documentation*, but to re-imagine how text and graphics might work together to support the *construction* of code. Ultimately, we envision rich diagrams that are graphical interfaces for *editing* code. (Unlike a post-ASCII tool for documentation, these interfaces would be transient for a single developer and would not need to be supported by all existing tools in use.) The diverse uses and forms of in situ diagramming revealed by our studies lay a foundation for realizing this future vision.

ACKNOWLEDGMENTS

Kevin Chang contributed to initial development of the codebook in section 4. We thank our interviewees who generously shared their time and expertise, and the anonymous reviewers for their thoughtful feedback. This material is based upon work supported by the National Science Foundation under Grant No. NSF IIS-1845900.

REFERENCES

[1] 2000. The Linux Kernel. See https://elixir.bootlin.com/linux/v6.6.8/source/arch/mips/include/asm/sn/klconfig.h#L217.

[2] 2006. The Linux Kernel. See https://elixir.bootlin.com/linux/v6.6.8/source/mm/filemap.c#L2300.

[3] 2014. The LLVM Compiler Infrastructure. See https://elixir.bootlin.com/llvm/llvmorg-17.0.6/source/llvm/lib/Target/AArch64/AArch64ConditionalCompares.cpp#L71.

[4] 2015. Chromium. See https://source.chromium.org/chromium/chromium/src/+/9ca452be82f2d6569a026729825ec747a05cd573:content/browser/renderer_host/render_frame_host_manager_unittest.cc;l=2639-2652.

[5] 2020. Tensorflow. See https://github.com/tensorflow/tensorflow/blob/5f3b48c8283266df2e17cfc2f1868397405cb5d0/tensorflow/compiler/xla/service/gpu/multi_output_fusion.h#L36-L92.

[6] 2021. The Linux Kernel. See https://elixir.bootlin.com/linux/v6.6.8/source/drivers/hwtracing/coresight/coresight-trbe.c#L298.

[7] 2022. The Linux Kernel. See https://elixir.bootlin.com/linux/v6.6.8/source/drivers/net/wireless/intel/iwlwifi/fw/api/phy-ctxt.h#L23.

[8] 2022. The Linux Kernel. See https://elixir.bootlin.com/linux/v6.6.8/source/tools/testing/selftests/bpf/prog_tests/xfrm_info.c#L3.

[9] 2022. The Linux Kernel. See https://elixir.bootlin.com/linux/v6.6.8/source/tools/testing/memblock/tests/alloc_api.c#L52.

[10] 2022. The LLVM Compiler Infrastructure. See https://elixir.bootlin.com/llvm/llvmorg-17.0.6/source/libc/src/__support/FPUtil/x86_64/FEnvImpl.h#L404.

[11] 2022. The LLVM Compiler Infrastructure. See https://elixir.bootlin.com/llvm/llvmorg-17.0.6/source/llvm/include/llvm/BinaryFormat/DXContainer.h#L28.

[12] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax To Textual Code. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA* (2020). https://doi.org/10.1145/3428290

[13] Oliver Arafat and Dirk Riehle. 2009. The commenting practice of open source. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 857–864. https://doi.org/10.1145/1639950.1640047

[14] Ian Arawjo. 2020. To Write Code: The Cultural Fabrication of Programming Notation and Practice. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI ’20)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3313831.3376731

[15] Ian Arawjo, Anthony DeArmas, Michael Roberts, Shrutarshi Basu, and Tapan S. Parikh. 2022. Notational Programming for Notebook Environments: A Case Study with Quantum Circuits. In *The 35th Annual ACM Symposium on User Interface Software and Technology, UIST 2022, Bend, OR, USA, 29 October 2022 - 2 November 2022*, Maneesh Agrawala, Jacob O. Wobbrock, Eytan Adar, and Vidya Setlur (Eds.). ACM, 62:1–62:20. https://doi.org/10.1145/3526113.3545619

[16] American Standards Association. 1963. American Standard Code for Information Interchange, X3.4-1963. See https://www.sensitiveresearch.com/Archive/CharCodeHist/X3.4-1963/index.html.

[17] Thomas Ball and Stephen G. Eick. 1996. Software Visualization In the Large. *Computer* 29, 4 (1996), 33–43. https://doi.org/10.1109/2.488299

[18] Sebastian Baltes and Stephan Diehl. 2017. Sketches and Diagrams in Practice. *CoRR* abs/1706.09172 (2017). arXiv:1706.09172 http://arxiv.org/abs/1706.09172

[19] Alex Bigelow, Steven Mark Drucker, Danyel Fisher, and Miriah D. Meyer. 2017. Iterating between Tools to Create and Edit Visualizations. *IEEE Trans. Vis. Comput. Graph.* 23, 1 (2017), 481–490. https://doi.org/10.1109/TVCG.2016.2598609

[20] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. https://doi.org/10.1191/1478088706qp063oa arXiv:https://www.tandfonline.com/doi/pdf/10.1191/1478088706qp063oa

[21] 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

[22] Johnny Chuah, Jiajie Zhang, and Todd R. Johnson. 2000. The Representational Effect in Complex Systems: A Distributed Representation Approach. https://api.semanticscholar.org/CorpusID:56413506

[23] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 341–354. https://doi.org/10.1145/2908080.2908103

[24] Unicode Consortium. 2022. *2.5 Encoding Forms*. Unicode Consortium, Chapter Chapter 2: General Structure, 33–38. https://www.unicode.org/versions/Unicode15.0.0/ch02.pdf

[25] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky (Eds.). 1993. *Watch What I Do: Programming by Demonstration.* MIT Press.

[26] Eduardo Santana de Almeida, Iftekhar Ahmed, and André van der Hoek. 2022. Let’s Go to the Whiteboard (Again): Perceptions from Software Architects on Whiteboard Architecture Meetings. *CoRR* abs/2210.16089 (2022). https://doi.org/10.48550/arXiv.2210.16089 arXiv:2210.16089

[27] Uri Dekel and James D. Herbsleb. 2007. Notation and representation in collaborative object-oriented design: an observational study. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 261–280. https://doi.org/10.1145/1297027.1297047

[28] Stephan Diehl. 2007. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software.* Springer. https://doi.org/10.1007/978-3-540-46505-8

[29] Robert Bruce Findler and PLT. 2023. Graphical Syntax. In *DrRacket: The Racket Programming Environment v8.10.* https://docs.racket-lang.org/drracket/Graphical_Syntax.html

[30] David Gesswein. 2004. ASR 33 Teletype Information. https://www.pdp8online.com/asr33/asr33.shtml [Accessed 14-Sep-2023].

[31] Camille Gobert and Michel Beaudouin-Lafon. 2023. Lorgnette: Creating Malleable Code Projections. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, UIST 2023, San Francisco, CA, USA, 29 October 2023- 1 November 2023*, Sean Follmer, Jeff Han, Jürgen Steimle, and Nathalie Henry Riche (Eds.). ACM, 71:1–71:16. https://doi.org/10.1145/3586183.3606817

[32] John Gruber. 2004. Markdown: Syntax. https://daringfireball.net/projects/markdown/syntax

[33] Philip J. Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *The 44th ACM Technical Symposium on Computer Science Education, SIGCSE 2013, Denver, CO, USA, March 6-9, 2013*, Tracy Camp, Paul T. Tymann, J. D. Dougherty, and Kris Nagel (Eds.). ACM, 579–584. https://doi.org/10.1145/2445196.2445368

[34] Devamardeep Hayatpur, Daniel Wigdor, and Haijun Xia. 2023. CrossCode: Multi-level Visualization of Program Execution. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, CHI 2023, Hamburg, Germany, April 23-28, 2023*, Albrecht Schmidt, Kaisa Väänänen, Tesh Goyal, Per Ola Kristensson, Anicia Peters, Stefanie Mueller, Julie R. Williamson, and Max L. Wilson (Eds.). ACM, 593:1–593:13. https://doi.org/10.1145/3544548.3581390

[35] Andrew Head, Amber Xie, and Marti A. Hearst. 2022. Math Augmentation: How Authors Enhance the Readability of Formulas Using Novel Visual Design Practices. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI ’22)*. Association for Computing Machinery, New York, NY, USA, Article 491, 18 pages. https://doi.org/10.1145/3491102.3501932

[36] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empir. Softw. Eng.* 25, 3 (2020), 2179–2217. https://doi.org/10.1007/S10664-019-09730-9

[37] Katherine E. Isaacs and Todd Gamblin. 2019. Preserving Command Line Workflow for a Package Management System Using ASCII DAG Visualization. *IEEE Transactions on Visualization and Computer Graphics* 25, 9 (2019), 2804–2820. https://doi.org/10.1109/TVCG.2018.2859974

[38] Bradley Kjell. 2020. Teletype Machines. https://chortle.ccsu.edu/AssemblyTutorial/Chapter-05/ass05_06.html [Accessed 14-Sep-2023].

[39] Donald Ervin Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111.

[40] Amy J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Human Factors in Computing Systems (CHI).*

[41] Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. 2020. The Design Space of Computational Notebooks: An Analysis of 60 Systems In Academia and Industry. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* https://doi.org/10.1109/VL/HCC50065.2020.9127201

[42] Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* (2009). http://www.aaai.org/ojs/index.php/aimagazine/article/view/2262

[43] Sorin Lerner. 2020. Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI ’20)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3313831.3376494

[44] H. Lieberman (Ed.). 2001. *Your Wish is My Command: Programming by Example.* Morgan Kaufmann Publishers Inc.

[45] 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

[46] Richard E. Mayer. 2014. *Cognitive Theory of Multimedia Learning* (2 ed.). Cambridge University Press, 43–71. https://doi.org/10.1017/CBO9781139547369.005

[47] United States of America Standards Institute. 1968. USA Standard Code for Information Interchange, X3.4-1968. See https://ia800800.us.archive.org/35/items/enf-ascii-1968-1970/Image070917151315.pdf.

[48] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Conference on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/3453483.3454059

[49] Josh Pollock, Catherine Mei, Grace Huang, Daniel Jackson, and Arvind Satyanarayan. 2023. Bluefish: A Relational Grammar of Graphics. *CoRR* abs/2307.00146 (2023). https://doi.org/10.48550/arXiv.2307.00146 arXiv:2307.00146

[50] Blaine A. Price, Ronald Baecker, and Ian S. Small. 1993. A Principled Taxonomy of Software Visualization. *J. Vis. Lang. Comput.* 4, 3 (1993), 211–266. https://doi.org/10.1006/jvlc.1993.1015

[51] M. Ragan-Kelley, F. Perez, B. Granger, T. Kluyver, P. Ivanov, J. Frederic, and M. Bussonnier. 2014. The Jupyter/IPython architecture. In *Fall Meeting Abstracts*. American Geophysical Union, Article H44D-07.

[52] Sawan Rai, Ramesh Chandra Belwal, and Atul Gupta. 2022. A Review on Source Code Documentation. *ACM Trans. Intell. Syst. Technol.* 13, 5 (2022), 84:1–84:44. https://doi.org/10.1145/3519312

[53] Pooja Rani, Arianna Blasi, Nataliia Stulova, Sebastiano Panichella, Alessandra Gorla, and Oscar Nierstrasz. 2023. A decade of code comment quality assessment: A systematic literature review. *J. Syst. Softw.* 195 (2023), 111515. https://doi.org/10.1016/J.JSS.2022.111515

[54] John Regehr. 2019. Explaining Code using ASCII Art. https://blog.regehr.org/archives/1653 [Accessed 02-Jun-2023].

[55] James Simpson and Michael Terry. 2011. Embedding Interface Sketches in Code. In *Proceedings of the 24th Annual ACM Symposium Adjunct on User Interface Software and Technology* (Santa Barbara, California, USA) *(UIST '11 Adjunct)*. Association for Computing Machinery, New York, NY, USA, 91–92. https://doi.org/10.1145/2046396.2046438

[56] David Canfield Smith. 1975. *Pygmalion: A Creative Programming Environment.* Ph. D. Dissertation. Stanford University.

[57] Ian Sommerville. 2011. Software engineering 9th. (2011).

[58] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 43–52. https://doi.org/10.1145/1858996.1859006

[59] Daniela Steidl, Benjamin Hummel, and Elmar Jürgens. 2013. Quality analysis of source code comments. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*. IEEE Computer Society, 83–92. https://doi.org/10.1109/ICPC.2013.6613836

[60] Masaki Suwa and Barbara Tversky. 2002. External Representations Contribute to the Dynamic Construction of Ideas. In *Diagrammatic Representation and Inference*, Mary Hegarty, Bernd Meyer, and N. Hari Narayanan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 341–343.

[61] Knut Sveidqvist. 2014. Mermaid. https://github.com/mermaid-js/mermaid

[62] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche (Eds.). IEEE Computer Society, 260–269. https://doi.org/10.1109/ICST.2012.106

[63] Edward R Tufte. 1991. Envisioning information. *Optometry and Vision Science* 68, 4 (1991), 322–324.

[64] Michael B. Twidale and David M. Nichols. 2005. Exploring Usability Discussions in Open Source Development. In *38th Hawaii International Conference on System Sciences (HICSS-38 2005), CD-ROM / Abstracts Proceedings, 3-6 January 2005, Big Island, HI, USA*. IEEE Computer Society. https://doi.org/10.1109/HICSS.2005.266

[65] Jörg von Engelhardt. 2002. *The Language of Graphics: A Framework for the Analysis of Syntax and Meaning in Maps, Charts and Diagrams.* Ph. D. Dissertation. University of Amsterdam.

[66] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, Yann-Gaël Guéhéneuc, Foutse Khomh, and Federica Sarro (Eds.). IEEE / ACM, 53–64. https://doi.org/10.1109/ICPC.2019.00019

[67] Koji Yatani, Eunyoung Chung, Carlos Jensen, and Khai N. Truong. 2009. Understanding How and Why Open Source Contributors Use Diagrams in the Development of Ubuntu. 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, 995–1004. https://doi.org/10.1145/1518701.1518853

A INTERVIEW QUESTIONS

Below is the series of questions asked in the semi-structured interview study:

(1) Can you provide me with a brief explanation of what the diagram is for?
(2) Why did you want to include the diagram as an ASCII drawing (as opposed to comments or external documentation)?
(3) What was your workflow for creating the diagram?
(4) Did you follow any guidelines or standards?
(5) What are the specific things about the code you are trying to explain with the diagram? How are those represented in the diagram?
(6) Who did you make the diagram for? For yourself (as reference)? For others?
(7) Have you made other diagrams in the past, was this diagram typical or unusual?
(8) Have you contributed to others' diagrams?
(9) Are there diagrams you've recently seen, which you found memorable?
(10) Are there other important aspects of these diagrams that are not reflected in our interview?

B CODEBASE SELECTION

Table 3 shows the code-bases we searched. They were filtered as follows: Git and FFmpeg did not have as substantial number of ASCII diagrams, and Gecko and Chromium had a similar domain, therefore only one of them (Chromium) was selected. These were the codebases we initially searched for, and we concluded our search having found a representative sample to study.

Table 3: Repository statistics of all searched code-bases.

Repo	LOC	Diagrams	LOC/Diagram
Chromium	56,697,921	428	132,471
Linux	35,415,763	1,386	25,552
LLVM	28,275,527	220	128,525
Tensorflow	6,618,475	122	54,249
FFmpeg	1,799,773	64	28,121
Gecko	52,194,803	579	90,146
Git	3,201,991	21	152,475