Terminals All the Way Down

Michael MacInnis
School of Computer Science
Carleton University
Ottawa, Canada
michaelpmacinnis@cmail.carleton.ca

Olga Baysal
School of Computer Science
Carleton University
Ottawa, Canada
olga.baysal@carleton.ca

Michele Lanza
REVEAL @ Software Institute
USI, Lugano
Lugano, Switzerland
michele.lanza@usi.ch

ABSTRACT

The terminal is a remarkably resilient interface for many programming activities. From its humble beginnings as a teletypewriter more than half a century ago, through video terminals like the VT100, to present-day windowed terminal emulators, it has remained a relevant and productive, albeit very gaunt, interface. This is in stark contrast with feature-rich integrated development environments (IDEs), which on top of their innate complexity allow for the creation of custom extensions. Indeed, researchers have been prolific in proposing innumerable, but often ignored, plug-ins.

We propose using inter-connected windowed terminal emulators as the foundation for a new type of distributed and language-agnostic development environment. By delegating the handling of a system's source code to a set of dedicated windowed terminal emulators we aim at complementing existing visual tools and leveraging the large body of existing command-line and terminal-based development tools. We present the architecture of the terminal-based development environment that we envision, outline our future implementation plans, and discuss how such an environment can be evaluated both in terms of its usefulness and usability.

KEYWORDS

Development Environment, Programming, Source Code, Terminal

ACM Reference Format:
Michael MacInnis, Olga Baysal, and Michele Lanza. 2022. Terminals All the Way Down. In *New Ideas and Emerging Results (ICSE-NIER'22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510455.3512784

1 INTRODUCTION

The terminal, and the command-line interface it presents, remains a relevant and often extremely productive interface, particularly on Unix and Unix-like systems, thanks to the Unix shell [10]. "Terminal emulators, or simply terminals, are used ubiquitously by developers" [16] and the terminal's ubiquity is not limited to professional settings. In education, as well, "the use of command line tools alone for teaching introductory programming is remarkably persistent" [24].

ICSE-NIER'22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9224-2/22/05
https://doi.org/10.1145/3510455.3512784

The terminal's command-line interface allows for typescript[1]-style interaction. "The typescript — an intermingling of textual commands and their output — originates with the scrolls of paper on teletypes" [26].

Teletypes or teletypewriters were eventually replaced by video terminals and those were in turn replaced by windowed terminal emulators. Opinions on the merits of windowed terminal emulators have always been divided:

> *The advent of windowed terminals has given each user what amounts to an array of teletypes, a limited and unimaginative use of the powers of bitmap displays and mice* [26].

> *The advent of windowing systems brought great advances in juxtaposability for all jobs where more than one task was performed concurrently. A notable improvement for many users was being able to run terminal emulations with windows, using one window for each thread under say X-windows and making a whole new way of working possible; in contrast, on the standard Unix platform without windows, all the threads had to be merged into one stream, making multiple concurrent sessions extremely hard [...] to manage* [20].

Using the full power of a graphical user interface to emulate a video terminal is unimaginative, but a video terminal and in particular the command-line interface it presents is already a powerful interface and, as stated above, simply having the ability to see more than one windowed terminal emulator at a time makes a whole new way of working possible.

Windowed terminal emulators, like the video terminals they emulate, are also not limited to supporting only typescript-style interactions. The cursor addressing extensions introduced by video display terminals and standardized as ANSI escape codes [5] allow *full screen* textual applications to be developed. Among these, the venerable `vi` editor which lives on as `vim` and continues to be popular with developers [4, 7, 8]. And windowed terminal emulators have continued to evolve, adding support for hyperlinks [21] and even raster graphics [19].

We propose a framework that adds two small layers of indirection in the hopes of allowing current windowed terminal emulators to serve as the foundation for a new type of distributed and language-agnostic development environment for Unix and Unix-like systems.

[1]Throughout this paper, the term *typescript* is used to mean the "intermingling of textual commands and their output" [26] and not as a reference to the strongly typed language built on top of JavaScript, developed and maintained by Microsoft, called TypeScript

In addition to building and experimenting with the possibilities presented by this new framework, we are investigating how best to evaluate these contributions. Ideally, we would like to conduct user studies with actual developers working on real problems.

2 COBBLERS, BAKERS, AND THE PERFECT MUSTARD

The cobbler always having the worst shoes is a popular idiom that exists in many similar forms. A common interpretation is that the cobbler is too busy making or fixing other people's shoes, and so the cobbler's own shoes are neglected. Many people believe that this is the case with software development. A popular misconception is that professional developers stick with antiquated approaches because they have been too busy to learn to use more productive tools. Joel Spolsky offers another interpretation when describing how bakers see things differently:

> *The whole concept of clean in the bakery was something you had to learn. To an outsider, it was impossible to walk in and judge whether the place was clean or not. An outsider would never think of looking at the inside surfaces of the dough rounder [...] to see if they had been scraped clean. An outsider would obsess over the fact that the old oven had discoloured panels, because those panels were huge. But a baker couldn't care less whether the paint on the outside of their oven was starting to turn a little yellow. The bread still tasted just as good. After two months in the bakery, you learned how to 'see' clean* [31].

The cobbler, like Spolsky's bakers, sees things differently. The cobbler's shoes may look worn out to those who do not spend a lot of time working with shoes but the cobbler is not distracted by superficial details. Similarly, many developers prefer the terminal, despite its gaunt interface, for its simplicity and reliability and because it provides access to tools that can be automated and scripted. We believe it is important to not be distracted by the superficial qualities of the terminal, like its age, and that by doing so and exploring a mostly unexplored area of the development environment ecosystem (see Figure 1), we can arrive at a solution that complements existing visual tools in addition to leveraging the large body of existing command-line and terminal-based development tools.

Our innate tribal tendencies often cause software development discussions to devolve into ridiculous debates: `emacs` vs. `vi`, tabs vs. spaces, the one true brace style vs. all the other *obviously* incorrect ways to format C-like languages, etc. Development environments are no exception but improving development environments is not a zero-sum game and treating it as such only serves to diminish the set of possible solutions. Gladwell, presenting the findings of Howard Moskowitz, says: "There is no perfect mustard or imperfect mustard. There are only different kinds of mustard [or spaghetti sauce, or coffee] that suit different kinds of people" [17].

Similarly, there is no perfect development environment. There are different kinds of development environments that suit different development styles and tasks.

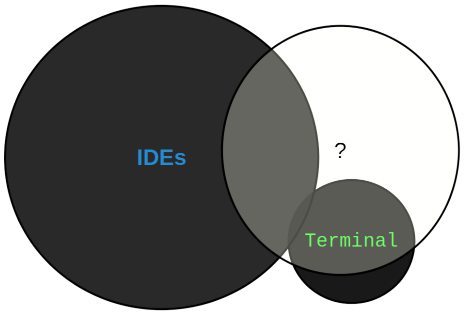
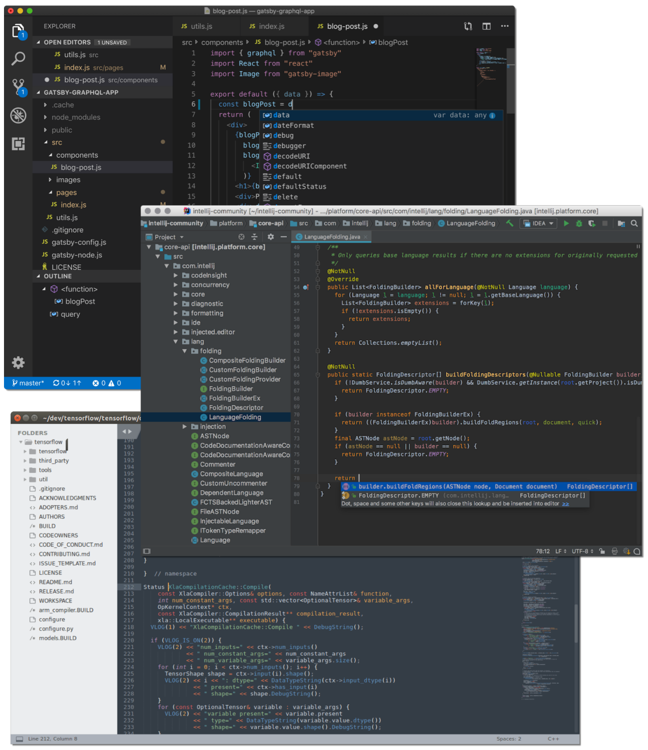


Figure 1: Development environments.

3 THE PROBLEM WITH IDES

The terminal can be a uniquely expressive interface but modern IDEs are also incredibly powerful. Many developers would balk at the idea of using anything else. Syntax highlighting, code completion, refactoring support, and interactive debugging are just a few of the features offered by modern IDEs [25].


blog-post.js — gatsby-graphql-app
utils.js
index.js
blog-post.js
src > components > blog-post.js > <function> > blogPost
intellij-community [~/intellij-community] - .../platform/core-api/src/com/intellij/lang/folding/LanguageFolding.java [intellij.platform.core]
LanguageFolding.java
78:12 LF UTF-8
Spaces: 2
C++

```
import { graphql } from "gatsby"
import React from "react"
import Image from "gatsby-image"

export default ({ data }) => {
  const blogPost = d
  return (
    <div>
      {blogP
        blog
        blog
          <I
        )}
      <h1>{b
      <div>P
```

```
data                var data: any
dateFormat
debug
debugger
decodeURI
decodeURIComponent
default
defaultStatus
delete
```

```
/**
 * Only queries base language results if there are no extensions for originally requested
 */
@NotNull
@Override
public List<FoldingBuilder> allForLanguage(@NotNull Language language) {
  for (Language l = language; l != null; l = l.getBaseLanguage()) {
    List<FoldingBuilder> extensions = forKey(l);
    if (!extensions.isEmpty()) {
      return extensions;
    }
  }
  return Collections.emptyList();
}

@NotNull
public static FoldingDescriptor[] buildFoldingDescriptors(@Nullable FoldingBuilder builder
  if (!DumbService.isDumbAware(builder) && DumbService.getInstance(root.getProject()).isDu
    return FoldingDescriptor.EMPTY;
  }

  if (builder instanceof FoldingBuilderEx) {
    return ((FoldingBuilderEx)builder).buildFoldRegions(root, document, quick);
  }
  final ASTNode astNode = root.getNode();
  if (astNode == null || builder == null) {
    return FoldingDescriptor.EMPTY;
  }

  return 
}
}
```

```
builder.buildFoldRegions(ASTNode node, Document document)   FoldingDescriptor[]
FoldingDescriptor.EMPTY (com.intellij.lang…     FoldingDescriptor[]
Dot, space and some other keys will also close this lookup and be inserted into editor >>
```

```
}

}  // namespace

Status XlaCompilationCache::Compile(
    const XlaCompiler::Options& options, const NameAttrList& function,
    int num_constant_args, const std::vector<OptionalTensor>& variable_args,
    OpKernelContext* ctx,
    const XlaCompiler::CompilationResult** compilation_result,
    xla::LocalExecutable** executable) {
  VLOG(1) << "XlaCompilationCache::Compile " << DebugString();

  if (VLOG_IS_ON(2)) {
    VLOG(2) << "num_inputs=" << ctx->num_inputs()
            << " num_constant_args=" << num_constant_args
            << " num_variable_args=" << variable_args.size();
    for (int i = 0; i < ctx->num_inputs(); i++) {
      TensorShape shape = ctx->input(i).shape();
      VLOG(2) << i << ": dtype=" << DataTypeString(ctx->input_dtype(i))
              << " present=" << ctx->has_input(i)
              << " shape=" << shape.DebugString();
    }
    for (const OptionalTensor& variable : variable_args) {
      VLOG(2) << "variable present=" << variable.present
              << " type=" << DataTypeString(variable.value.dtype())
              << " shape=" << variable.value.shape().DebugString();
```

Figure 2: Visual Studio Code, IntelliJ, and Sublime Text.

The problem with IDEs, at least from a tool developer's perspective, is the *total control* approach they take and how this results in increasingly large and complicated code bases. Popular IDEs also share little, if any, code. Researchers building software development tools as IDE plug-ins must expend a significant amount of effort which has to be repeated for each targeted IDE "as the tools have to be ported to [each] IDE to benefit maximally from it" [29]. This is aggravated by a tendency that IDEs have to fall out of fashion. The complexity of their code bases and plug-in ecosystems seem to require a minimum popularity to remain viable and relevant.

Despite having distinct, increasingly large, and complicated code bases, IDEs have converged, over the last forty years, on "a 'bento box' design: the screen is partitioned into rectangular areas that contain editors (e.g., code editors, user interface designers), navigators (e.g., project viewers, class viewers), and tool output (e.g., search results, compilation errors)" [14]. This "bento box" design (see Figure 2) does not exploit the possibilities that multiple monitors present nor is it helpful in allowing developers "to form and exploit spatial memory" [14].

4 PROPOSAL

Writing code, despite attempts at richer representations, is mostly handling text. Text "is a universal interface" [28], and although it may not be the best way to represent programs it is better than "all those other forms that have been tried from time to time" [11]. We believe that current windowed terminal emulators are excellent at handling text and powerful enough to serve as the foundation for a new type of distributed and language-agnostic development environment for Unix and Unix-like systems. To investigate this we propose building a system that adds two layers of indirection (see Figure 3).

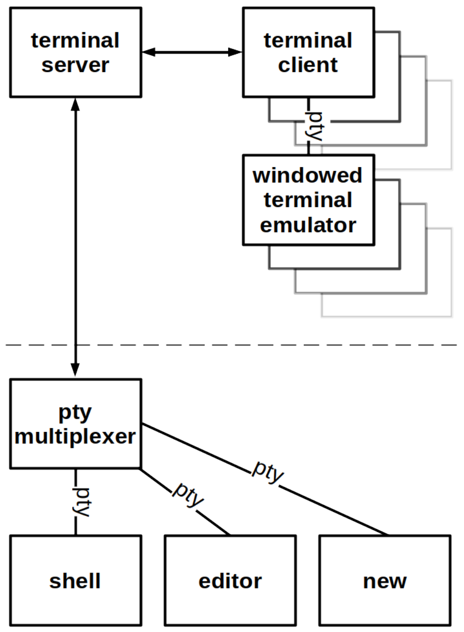


Figure 3: Proposed Framework.

The first layer of indirection we call the *terminal server*. The terminal server exists locally and manages all windowed terminal emulator instances on the local machine. It knows which windowed terminal emulator instances exist and what they are doing. The terminal server serves multiple windowed terminal clients. The windowed terminal client, or just *terminal client*, allows us to avoid having to implement a competitive terminal emulator from scratch. Instead we can rely on any of the many terminal emulators that already exist. Instead of the windowed terminal emulator communicating with a program (through a pty) it communicates with the terminal client (through a pty) which communicates with the terminal server which manages all sessions. These sessions rely on the second layer of indirection, which we call a *pty multiplexer*.

The pty multiplexer exists locally and on all remote machines and inside any and all containers we expect to be able to use as part of our distributed development environment. This layer of indirection is not a new idea. The pty multiplexer is similar to `screen` [3] or `tmux` [9] but in addition to handling multiple pty sessions it multiplexes output and demultiplexes input. By cooperating with the terminal server, input is sent to the appropriate program and output to the appropriate local window. We are aiming for the pty multiplexer to be small enough and well contained enough that installing it would be similar to installing `screen` or `tmux`. One pty multiplexer may end up indirectly launching other pty multiplexers over SSH on remote machines or in containers. The pty multiplexer instances form a tree and work together to route input and output.

All communication between components will be encoded using an ANSI escape sequence [5], referred to by ECMA-48 as a "Privacy Message" [1]. Privacy Messages are ignored by all `xterm`-like windowed terminal emulators [2]. We envision the pty mutliplexer signalling its ability to handle a multiplexed data stream with an advertisement encoded in this way. Encoding this advertisement as a Privacy Message will ensure that it is harmless in the case that the receiver does not support multiplexing which will allow the pty multiplexer to fallback to behaving as a transparent shim.

This infrastructure will allow us to preserve the illusion that breaks when the terminal is used to access remote or containerized systems. We want to make remote and containerized development more seamless by *baking in* the idea that all development does not occur on the local machine. The terminal server, co-operating with the pty multiplexer locally, on remote systems, or in a containers, will be able to launch programs on those systems with their own local windows.

5 USE CASES

Even in an embryonic state we believe this framework will be useful for remote and containerized development.

Containerized Development. Launching a terminal inside a container can be as simple as typing `docker run`. Launching additional terminals, however, is more cumbersome. Additional steps are required to "connect" to the container again. Our framework will solve this problem and only require the pty multipexer to be installed and configured as the entrypoint for an image.

Remote Containerized Development. Our framework will require no additional configuration to allow multiple terminals to be launched inside a remote container with the pty multiplexer installed and configured as the entrypoint. If, additionally, we want to easily launch multiple terminals on the remote server we can configure and install the pty mutliplexer on the remote server but even without this our framework will happily multiplex terminals inside the remote container transparently through the SSH connection to the remote server.

6 FUTURE PLANS

We propose developing or extending a terminal-based editor so that it can cooperate with our framework. Our goal is to be able to have the output of these windows affect each other. We see many common IDE features as simply the juxtaposition of one or more textual windows with the ability to control one textual window (cursor position, colour, highlighting or other annotations) with the output from another window. Amazingly, there is already a convention most command-line tools follow when referring to code even if they do not know it by name — "code addresses" [13].

There are many modern command-line and terminal-based tools that offer much more than simple typescript-style interaction [12, 15, 18]. We are excited by the possibilities offered by a framework that will allow these tools to communicate and co-operate. The framework we propose here is only the first step.

7 EVALUATION PLAN

Our aim is to build a framework that is useful on its own while significantly reducing the effort required to build subsequent development tools. A key part of this will be validating our assumption that windowed terminal emulators are powerful enough to serve as a foundation. To do this we plan to start by not only evaluating how easily the framework allows us to mimic common useful IDE features but also investigating the opportunities it presents to support features that are not commonly found in popular IDEs. We believe that this is a fairly unique time in the history of software development, and that these efforts will be aided by the considerable amount of consolidation not only in popular languages and tools but in how we approach software development itself [22].

If our assumptions about windowed terminal emulators being powerful enough are correct, we then plan to use this framework "for our daily work, not just as a research tool. [This will force] us to address shortcomings as they arise and to adapt the system to solve our problems" [27]. We intend to leverage existing open-source tools as much as possible and also release components of our framework as open source as soon as they reach a minimum viable state. Our goal is to attract other like minded cobblers, bakers, developers. We hope that there will be enough interest to attract a sufficient number of volunteers to allow evaluation through interviews and questionnaires and that these will allow us uncover any lingering issues and unnecessary barriers to adoption. Finally, we plan to perform a comparative evaluation between our framework and other popular tools [32]. As previously stated, in addition to building and experimenting with the possibilities presented by this new framework, we are investigating how best to evaluate these contributions. Our investigation into these methods is ongoing.

Our target audience is anyone who needs to edit text files. Particularly in the initial stages, we feel that our approach will be suited to small, emerging, and domain-specific languages that do not yet, or may never, have good IDE support. Developers are also only part of our target audience. Nebulous definitions of dev-ops aside, the basic idea of dev-ops is that developers and operators, or administrators, perform many of the same tasks and, we believe, can be served by the same tools.

In later stages we hope that the consolidation we see in tools and approaches along with projects like the Language Server Protocol [6] which decouple language services from editors will allow us to achieve feature parity with existing IDEs. Our hope is that by delegating text handling to windowed terminal emulators, window management to the window manager, and access to remote and containerized systems to existing terminal-based tools, our approach will be easier to extend than existing IDEs while also having a significantly lower total complexity.

"A [tool's] got to know [its] limitations"
– Dirty Harry

Our ultimate goal is not to promote text and the terminal interface above all other interfaces but to "infiltrate existing development environments and complement the existing functionalities" [23]. We believe that tools including, but not limited to, "software visualization should have a symbiotic relationship with the practice of code reading by pointing the viewer to the location in the system where [they] should read and/or modify the code" [23].

8 CONCLUSION

"From a terminal, you SSH to a VM and get what?"
"A terminal."
"And then you docker exec and get what?"
"Another terminal. It's terminals all the way down."

The terminal is a remarkably resilient interface. In its current incarnation(s), as the many and various windowed terminal emulators, the terminal has evolved beyond supporting a typescript-style of interaction and into "a general support environment for text-based programs" [30]. Meanwhile, IDEs have converged on a de facto standard, "bento box", design which does not take full advantage of increasingly common multiple monitor environments nor help developers "form and exploit spatial memory" [14].

Worse, the significant effort expended by researchers to develop plug-ins for IDEs is often ignored and can be largely wasted if and when the targeted IDE falls out of fashion.

We suspect that current windowed terminal emulators are powerful enough to serve as the foundation for a new type of distributed and language-agnostic development environment for Unix and Unix-like systems. We hope that the persistence and adaptability of the terminal will mean that tools developed for our framework will be more resilient and that our framework will lower the bar for software development tool builders and allow for more and easier experimentation.

Acknowledgements

MacInnis and Baysal gratefully acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN-2021-03809. Lanza gratefully acknowledges the financial support of the Swiss National Science Foundation (SNF) for the project "INSTINCT" (SNF Project No. 190113).

REFERENCES

[1] 1991. ECMA-48: Control functions for coded character sets. https://www.ecma-international.org/publications-and-standards/standards/ecma-48/.
[2] 2005. Xterm Control Sequences. https://www.xfree86.org/current/ctlseqs.html.
[3] 2016. GNU Screen. https://www.gnu.org/software/screen/.
[4] 2019. State of Haskell Survey Results. https://taylor.fausak.me/2019/11/16/haskell-survey-results/.
[5] 2020. All Escape Codes. https://bjh21.me.uk/all-escapes/all-escapes.xhtml.
[6] 2020. Language Server Protocol Specification - 3.16. https://microsoft.github.io/language-server-protocol/specification.
[7] 2021. Go Developer Survey Results. https://go.dev/blog/survey2020-results.
[8] 2021. Stack Overflow Developer Survey. https://insights.stackoverflow.com/survey/2021.
[9] 2021. tmux. https://github.com/tmux/tmux.
[10] Jon Bentley, Don Knuth, and Doug McIlroy. 1986. Programming Pearls: A Literate Program. *Communications of the ACM* 29, 6 (June 1986), 471–483.
[11] Winston Churchill. 1947. The Worst Form of Government. https://winstonchurchill.org/resources/quotes/the-worst-form-of-government/.
[12] Maxime Coste. 2021. Kakoune. https://kakoune.org/.
[13] Russ Cox. 2021. Code Addresses. https://pkg.go.dev/rsc.io/rf#hdr-Code_addresses.
[14] Robert DeLine and Kael Rowan. 2010. Code Canvas: Zooming Towards better Development Environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. 207–210.
[15] Brendan Falk. 2021. Fig. https://fig.io/.
[16] Ishaan Gandhi and Anshula Gandhi. 2020. Lightening the Cognitive Load of Shell Programming. http://reports-archive.adm.cs.cmu.edu/anon/isr2020/CMU-ISR-20-115B.pdf. *PLATEAU 2020* (Nov. 2020).
[17] Malcolm Gladwell. 2004. Choice, Happiness and Spaghetti Sauce. https://www.ted.com/talks/malcolm_gladwell_choice_happiness_and_spaghetti_sauce.
[18] Kovid Goyal. 2021. kitty. https://sw.kovidgoyal.net/kitty.
[19] Kovid Goyal. 2021. Terminal Graphics Protocol. https://sw.kovidgoyal.net/kitty/graphics-protocol/.
[20] Thomas Green and Alan Blackwell. 1998. Cognitive Dimensions of Information Artefacts: a tutorial. https://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf.
[21] Egmont Koblinger. 2020. Hyperlinks in Terminal Emulators. https://gist.github.com/egmontkob/eb114294efbcd5adb1944c9f3cb5feda.
[22] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. 2016. Code Review Quality: How Developers See It. In *Proceedings of the 38th ACM/IEEE International Conference on Software Engineering*. 1028–1038.
[23] Michele Lanza. 2003. Program Visualization Support for Highly Iterative Development Environments.. In *Proceedings of the 2nd International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT*. 67–72.
[24] Lauri Malmi, Ian Utting, and Amy J. Ko. 2019. *Tools and Environments*. Cambridge University Press.
[25] Gail C. Murphy, Mik Kersten, and Leah Findlater. 2006. How are Java software developers using the Elipse IDE? *IEEE Software* 23 (2006), 76–83.
[26] Rob Pike. 1994. Acme: A User Interface for Programmers.. In *USENIX Winter*. 223–234.
[27] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. 1995. Plan 9 from Bell Labs. *Computing Systems, Summer* 8, 3 (1995), 221–254.
[28] Eric Steven Raymond. 2003. *The Art of Unix Programming*. Addison-Wesley.
[29] Romain Robbes and Michele Lanza. 2007. *Towards Change-Aware Development Tools*. Technical Report. Università della Svizzera italiana.
[30] Chris Siebenmann. 2021. The xterm terminal emulator can do a lot more than just display text. https://utcc.utoronto.ca/~cks/space/blog/unix/XTermQuiteSophisticated.
[31] Joel Spolsky. 2005. Making Wrong Code Look Wrong. https://www.joelonsoftware.com/2005/05/11/making-wrong-code-look-wrong/.
[32] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer Science & Business Media.