Exploring Exploratory Programming

Mary Beth Kery
Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213
mkery@cs.cmu.edu

Brad A. Myers
Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213
bam@cs.cmu.edu

***Abstract*— In open-ended tasks where a program's behavior cannot be specified in advance, *exploratory programming* is a key practice in which programmers actively experiment with different possibilities using code. Exploratory programming is highly relevant today to a variety of professional and end-user programmer domains, including prototyping, learning through play, digital art, and data science. However, prior research has largely lacked clarity on what exploratory programming is, and what behaviors are characteristic of this practice. Drawing on this data and prior literature, we provide an organized description of what exploratory programming has meant historically and a framework of four dimensions for studying exploratory programming tasks: (1) applications, (2) required code quality, (3) ease or difficulty of exploration, and (4) the exploratory process. This provides a basis for better analyzing tool support for exploratory programming.**

Keywords—Exploratory Programming; Creativity Support; End-user programming.

I. INTRODUCTION

In coding and in general, *exploration* is a basic human strategy for gaining new understanding about a space of ideas. In a seminal paper on organizational learning, March [1] argued for a healthy balance between *exploitation*, the use of familiar knowledge, and *exploration*:

"*Exploration includes things captured by terms such as search, variation, risk taking, experimentation, play, flexibility, discovery, innovation. Exploitation includes such things as refinement, choice, production, efficiency, selection, implementation, execution.... Maintaining an appropriate balance between exploration and exploitation is a primary factor in system survival and prosperity.*" [1]

The term "Exploratory Programming" was first popularized in a 1983 paper by Beau Shiel, a manager at Xerox's AI Systems, who struggled with applying rigid software development lifecycles of the time to experimental AI code [2]. The trouble was that something so experimental as an AI system could not be fully specified up-front: "*no amount of interrogation of the client or paper exercises will answer these questions; one just has to try some designs to see what work*s" [2]. Before taking the long and expensive step of building software, it is generally recommended to rapidly iterate ideas through discussions and paper prototypes [3]. A simple paper-prototype of a user-interface, for example, can be shown to users to cheaply test and identify problems in the proposed design [3]. However, it is not always feasible to test an idea on paper. When an idea relies on processing data, or on computing a complex visual, sound, or motion effect, these behaviors can be difficult or even impossible to simulate in "low-fi" prototyping mediums like paper or a whiteboard. Exploratory programming fills the crucial role of the medium when prototyping must occur in code. This practice is key for situations where key attributes of exploration: "*flexibility, discovery, and innovation*" [1] are needed to understand how the program should behave.

At first glance, exploratory programming may seem difficult to separate from normal programming. Typical programming requires some experimentation and some creative problem-solving to reach a goal [4][5]. However, the practice of designing the goal *at the same time* as experimenting in code is a defining feature of exploratory programming. We extend Shiel's definition: "*the conscious intertwining of system design and implementation*" [2] to define exploratory programming as a task with two properties:

(Property 1) The programmer writes code as a medium to prototype or experiment with different ideas.

(Property 2) The programmer is not just attempting to engineer working code to match a specification. The goal is open-ended, and evolves through the process of programming.

To further gain an intuition for the boundaries of exploratory programming, consider debugging. Debugging often involves hypothesis testing and experimental code edits as a programmer tries to make sense of an error [5]. Debugging thus meets Property 1. However, the broad space of debugging is not exploratory programming. Consider that a programmer knows how their program should be behaving, and finds an error. Through problem solving the programmer finds the erroneous lines of code, and through further experimentation, develops a fix. We do not label this as exploratory programming because at no point is the programmer re-evaluating their system design or how their program *should* behave (Property 2). Later, the programmer might decide that the system should perhaps behave in a different way, and then might engage in exploratory programming to try out new possible behaviors.

Our observations are partly motivated by our prior studies of exploratory programming [6], where we interviewed 10 researchers (2 female, 8 male) who self-identified as doing exploratory programming, followed by a broader online survey which received 60 responses from data scientists. Briefly, participants emphasized the trial-and-error nature of their exploratory programming work, and they reported using a variety of simple methods such as duplicating files, duplicating code snippets, or commenting out code snippets in order to keep

This research was funded by NSF grant IIS-1314356. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the US Government.

978-1-5386-0443-4/17/$31.00 ©2017 IEEE

multiple versions of the same code visible at once. Full results are in [6].

In this paper we seek to clarify and deepen understanding of exploratory programming, by synthesizing evidence of how exploratory programming is used in the wild and how exploration affects a programmer's behavior. Today, we see highly exploratory code tasks, such as data science, learning through play, and computational art and design which need appropriate tool support, not only for professional developers but also for the broader end-user developer (EUD) audience. For example, programmers have a need to work with alternative versions and variants of their exploratory code [7] [6]. Our aim is to fuel future research in exploratory programming by grounding the practice in 5 characteristics. These characteristics are distilled from data from our own prior studies [6], as well as prior literature. These are discussed at length in later sections, along with related terms:

1. *Needs for Exploration*: Certain scenarios call for exploration. This includes learning how to do an unfamiliar task, working on creative tasks, or working on hard tasks where the means to achieving a goal is not apparent without experimentation.
2. *Code Quality Tradeoffs*: Exploratory programming emphasizes iteration on the ideas behind the code, so code quality is often deemphasized during exploration to allow for faster iteration. When the programmer reaches a final solution, they may then polish and refine the code.
3. *Ease or Difficulty of Exploration*: Usability factors in the languages, libraries, and tools that the programmers use affect a programmer's ability to rapidly prototype. From Green's cognitive dimensions [8], we identify that high Closeness of Mapping and low Viscosity are particularly helpful to exploration.
4. *Exploration Process*: Exploration is instantiated as repeated changes to parameters, input data, or certain regions of code over time. This process also includes backtracking and comparing current code to past attempts to decide what ideas to experiment with next.
5. *Group or Individual Exploration:* Exploratory programming is often done on an individual basis, but can become highly convoluted when a team needs to coordinate their experimentation.

II. Related Terms

Recently, Bergström and Blackwell [9] discussed a number of programming practices, framing code as a medium with a great many other uses than typical software engineering work. Some of these creative practices they discuss: bricolage, tinkering, sketching, live coding, and hacking, we consider to be subset of exploratory programming. What makes "exploratory programming" a useful framing across many programming practices? From bricolage to hacking, as well as other practices we discuss below, a common trait is that the programmer's goal is at least to some degree creative and open-ended. By examining exploratory programming, we can study the consequences on programmer behavior of working towards an open-ended goal, as well as tools that may benefit all of these practices that rely on goal exploration. Below we define several other relevant terms:

Opportunistic programming: As defined by Brandt et al. [10]: "*Programmers build software from scratch using high-level tools, often add new functionality via copy-and-paste, [and] iterate more rapidly than in traditional development….*" Past work on opportunistic programming has focused on web foraging as a way that programmers rapidly iterate on their ideas in code, by largely patching together a program from online examples. As the programmer is often exploring different possibilities of their program, Opportunistic Programming can be considered a subset of exploratory programming. However, there are behaviors specific to this practice that do not strictly generalize to the broad range of all exploratory programming tasks. Patching-together example code is one exploratory programming tactic that may be specific to Opportunistic Programming.

Debugging into existence: Rosson and Carroll [11] observed that Smalltalk programmers write partial code and run it, so that the resulting errors could point them towards where to improve the program. This style of programming is highly incremental and can be used for exploratory programming, if the programmer has an open-ended goal.

Rapid Prototyping: "*The rapid production of a prototype*" [12] is commonly used in iterative design to create and test a number of different design possibilities early on. Note that rapid prototyping may or may not involve programming, and the user may have a specific goal and design in mind. Rapid prototyping in *code* is exploratory programming if the programmer is exploring a variety of designs/goals rather than simply iterating on a single design.

III. Characteristics of Exploratory Programming

A. Needs for Exploratory Programming

Exploratory programming has been observed and purposefully supported in a wide variety of applications which fundamentally require exploration, including:

- Learning programming through play: Environments for children such as Alice [13] and Scratch [14] encourage creating stories through exploratory programming.
- Digital art and music: Digital art written in languages like Processing is created through experimentation with code as a creative medium [15][16]. Environments for generative music, often using live coding, involve impromptu exploration of sounds through code [17].
- Data Science: Tasks like data analysis are often done in code and are exploratory [18]. Other tasks, like modeling or building a machine-learning model, can also take extensive exploration and iteration in code [19].
- Software Engineering: Exploratory programming has been found in programmer's backtracking, where the programmer tries and retries different alternatives using commenting or undo commands while trying to deter-

mine an appropriate algorithm [20] or figuring out how an API should be used [21].

B. Code Quality Tradeoffs

Programmers frequently need to make a cost/benefit tradeoff between producing high-quality code, and spending their time and effort on quick ideation. Historically, this has meant that exploratory programming is associated with rough code: "*Exploratory programming techniques encourage code that is hard to read. It is tempting to make fix after fix to a piece of code until it is impossible to understand. Hence, rewriting code is essential for producing reusable code*" [22].

In our interview study [6], even participants who had formal software engineering backgrounds leaned towards messier practices when exploring:

"I know how to write code. And I know that I could write functions to reuse functions and I could try to modularize things better, and sometimes I just don't care because why am I going to put effort in that if I'm not going to use it again?"

This sentiment was common, but certainly prone to individual differences. When participants were writing code to be part of a maintained software system, or were simply more meticulous, they sometimes strived to follow good coding practices. However, all participants mentioned some way in which they preferred to reduce engineering effort while writing exploratory code, whether ignoring modularity, skipping documentation, or avoiding software version control. Some participants felt they wouldn't use this code long-term. Others did not want to invest too much time on code that may turn out to be a bad avenue that would need to be re-written later anyway.

In a study of game developers, Murphy-Hill et al. had similar findings [23]. Game development often requires exploratory programming due to evolving requirements and the creative nature of building a game. Even though subjects were professional game developers, they expressed this quality trade-off: "*there is a tradeoff between improving maintainability early and the likelihood that this effort will result in waste because the game will not be a success*" [23]. Programmers may choose low-investment in their code quality due to time pressures and risks of their immediate work later being thrown out for a new idea. Yet exploratory code faces the same problems with bugs and logic errors as any program. Poor quality exploratory code can lead to serious problems, such as incorrect data analyses o invalid scientific findings [24]. Exploration is a challenging area in which to support code quality.

C. Ease or Difficulty of Exploration

In the course of exploration, a programmer may need to significantly edit the design of their program to try a new approach. All programing tasks, including exploratory changes, can be made easier or more difficult by the tools available to the programmer. Here we use Green's cognitive dimensions of programming languages [8] to discuss particular usability features that support exploration.

First, languages that cost a programmer more time and effort to express a single idea will slow down iteration. This relates to Green's notion of *Diffuseness* versus *Terseness* of a language, which counts how many code symbols are required to express an idea. For exploratory programming, high-level languages and libraries can be extremely helpful to provide a higher-level vocabulary to make code more succinct. For instance, instead of coding the details of a computer vision algorithm to detect contours in an image, a programmer can simply use a one-line library call such as `find_contours()` [25].

Green's *Closeness of Mapping* is how directly a concept in the user's task domain maps to a code representation [8]. Good closeness of mapping for exploratory programming will favor higher-level abstractions for all parts of a program that are not the programmer's primary focus. For example, `find_contours()` allows a programmer to use computer vision without needing to understand its low-level algorithms. Green called side tasks that *only* relate to programming, such as explicit memory management, "programming games" [8]. We expand the "programming games" notion to encompass *any* supporting domain that a programmer may want to make use of without having to care about the details of that domain. The overall goal of close mapping and terse code is to help an exploratory programmer spend more time and effort focused on their ideas rather than the details required to enable them. Here, exploratory programming has certain overlap with opportunistic programming, and is farther from standard software development which prioritizes engineering goals such as efficiency of execution, flexibility, maintainability, and robustness over easy implementation. Close mapping also makes exploration more accessible to novices and end-user programmers who may be missing certain skills. For experts, Closeness of Mapping also implies that abstraction should be as fine or coarse-grain as is reasonable for the task. For example, if a domain expert is exploring new algorithms for computer vision, a very low level of abstraction may be appropriate.

Viscosity [8] refers to how easy or difficult it is to make a change in a program once it is written: "*programmers will choose their style of working according to the particular combination of information structure and editing tools. If the system is viscous, they will attempt to avoid local changes and will therefore avoid exploratory programming*" [4]. To avoid viscosity, highly modular programs may help programmers explore a single component without needing to propagate changes across the entire system to make that change run. High viscosity can also lead to errors. For example, one of our interview participants [6] discussed errors that resulted from reusing the same block of code for multiple experiments. Editing the code for one exploration made it difficult to maintain or recover an earlier exploration. Here, viscosity must take into account how easy it is both to create a change and to revert it. Exploratory programming environments should lower the risk of making exploratory edits by providing clear ways to return to prior versions. A simple undo is not sufficient when programmers make changes over time that they would like to only partially revert [20]. Prior work has found end-user programmers, even in visual programming, benefit from more sophisticated version control support to help with reverting [26].

D. Exploration Process

The *process* by which exploratory programming is done can be characterized by backtracking, the scale and duration of the changes, and history.

- *Backtracking*

Yoon et al. [20] examined 1,460 hours of programming log data from 21 programmers and identified evidence of exploratory programming when a programmer edited and backtracked the same piece of code over and over between runs. Following Yoon's convention, we identify an instance of exploration as two or more edit-run cycles that are close in time and affect the same code. Segmenting exploratory changes to code by runs may often be appropriate, as a programmer is experimenting with changes to the program, and must typically run the code in order to see the effect of those changes.

- *Exploration scale*

Identifying the scale of an exploratory section of code is helpful, as scale affects how a programmer will interact with the variants of that code. In conventional software version control tools, a set of changes on a source file is captured at the file level. However, in a practical exploratory experiment, the programmer may be manipulating a much more focused set of code than the entire file. At the smallest scale, a well-documented behavior is "tuning" a single variable or parameter to many different values to observe the effect [20][27]. An artist, for instance, may change a parameter in a program that generates a complex geometric shape [28]. At the next largest scale, a programmer may be rapidly iterating variations of a particular function. For instance, one of our interview participants created two copies of the same function to try two different approaches to an analysis and still keep both approaches. At the next larger scale are loose snippets of code that span multiple functions, or are not contained by a function at all. For example, one interview participant ran a different "configuration" of their file by commenting out certain lines across their file and uncommenting others to change what analysis was performed. Finally, exploration may also occur at the file level.

- *Exploration duration*

As a programmer shifts among different tasks, the exploration scale may vary not only by code size, but also by time spent. Exploration may be very transient or very long-term, depending on the task. For instance, if a programmer is changing the color and size of a button, this may be only of concern while the programmer is deciding which color and size is liked best. This contrasts with cases of building computational models where the programmer is involved in exploratory programming for weeks to months and must keep track of many explorations that make up the components of their model [19]. Relating to Green's cognitive dimension of *hard mental operations*, the burden on a programmer will be greater for keeping track of many attempts over a long period of time than keeping track of fewer code variations over a short period of time [8]. All of our interview participants used notes, code comments, or recorded output as external memory aides. However, in line with Code Quality (Characteristic B), most noted a high cost in effort to keep these manual records up-to-date.

- *Using Exploratory History*

In software engineering, history involves byproducts of the process such as code versions, commit messages, and issue logs, as well as the code changes themselves. Prior work has found that software engineers typically use code history to understand a change or bug [29]. On the exploratory end of the spectrum, exploratory programmers may use history for typical software engineering needs, but they also commonly use code history as a record of their experimentation. Abundant evidence for this is seen in scientific computing [30]. Whereas a software engineer might ask a code-centered question like: "In which version did this code appear?" [31], an exploratory programmer may frame their question around an experiment: "When I tried a RandomForest algorithm, how did that effect my model's accuracy?" or "When I went through this loop 3 times, how did this affect the character's motion?". Using history for this kind of question and answer was mentioned by several interview participants. To facilitate answering these questions, a programmer during exploration may need a variety of artifacts that can help them understand a past decision and its effect. This includes past versions of images, notes, variable values, parameters, and graphs, along with the code. In interviews, we saw participants keeping versions of code, jotting notes on ideas, and keeping versions of outputs. Of survey participants, 72% manually copied versions of their files and 3/10 interviewees and 52% of survey participants used a software version control tool.

E. Sharing and Group Exploration

Group exploration on the same code can be challenging because the kind of informal coding practices that appear in exploratory programming do not lend well to clear code. Due to programmers' reluctance to keep up-to-date notes during exploratory tasks, keeping a shared understanding of an exploration's progress across a team can be difficult. Sharing and group exploration also affects the viscosity of code edits, because where more than one person is making exploratory changes to the source code, this can easily lead to conflicting changes. Future work is needed to better understand the behaviors of exploratory programming in groups and how to best coordinate exploration among a team of people.

CONCLUSIONS AND IMPLICATIONS

Exploratory programming is a practice and a lens we can use to better understand and support creative and open-ended programming tasks. Although exploratory programming is prevalent across many applications today, there is currently a lack of tool support for experimentation, including a lack of support for recording and sensemaking of exploration history, and a lack of support for exploration by groups of people. As greater numbers of people grow up learning programming, it is particularly important that we provide tools not just to learn programming in the first place, but tools which allow programmers to use code in their own domain-specific practices as a creative, handy tool.

REFERENCES

[1] J. G. March, "Exploration and exploitation in organizational learning," *Organ. Sci.*, vol. 2, no. 1, pp. 71–87, 1991.

[2] B. Sheil, *Power tools for programmers*. Morgan Kaufmann Publishers Inc., 1986.

[3] B. Buxton, *Sketching user experiences: getting the design right and the right design*. Morgan Kaufmann, 2010.

[4] T. Green, "Programming Languages as Information Structures," in *Psychology of Programming*, J. M. Hoc, T. R. . Green, R.

Samurcay, and D. J. Gilmore, Eds. Academic Press, 1990, pp. 117–137.

[5] A. Ko and B. Myers, "Debugging reinvented," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, 2008, pp. 301–310.

[6] M. B. Kery, A. Horvath, and B. Myers, "Variolite: Supporting Exploratory Programming by Data Scientists," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2017.

[7] B. Hartmann, L. Yu, A. Allison, Y. Yang, and S. R. Klemmer, "Design as exploration: creating interface alternatives through parallel authoring and runtime tuning," in *Proceedings of the 21st annual ACM symposium on User interface software and technology*, 2008, pp. 91–100.

[8] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *J. Vis. Lang. Comput.*, vol. 7, no. 2, pp. 131–174, 1996.

[9] I. Bergström and A. F. Blackwell, "The practices of programming," in *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*, 2016, pp. 190–198.

[10] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer, "Opportunistic programming: How rapid ideation and prototyping occur in practice," in *Proceedings of the 4th international workshop on End-user software engineering*, 2008, pp. 1–5.

[11] M. B. Rosson and J. M. Carroll, "Active programming strategies in reuse," in *European Conference on Object-Oriented Programming*, 1993, pp. 4–20.

[12] "Oxford English Dictionary." .

[13] C. Kelleher, R. Pausch, and S. Kiesler, "Storytelling alice motivates middle school girls to learn computer programming," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007, pp. 1455–1464.

[14] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and others, "Scratch: programming for all," *Commun. ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[15] C. Reas and B. Fry, *Processing: a programming handbook for visual designers and artists*, no. 6812. Mit Press, 2007.

[16] N. Montfort, *Exploratory Programming for the Arts and Humanities*. MIT Press, 2016.

[17] A. R. Brown and A. Sorensen, "Interacting with generative music through live coding," *Contemp. Music Rev.*, vol. 28, no. 1, pp. 17–29, 2009.

[18] J. W. Tukey, "Exploratory Data Analysis," *Analysis*, vol. 2, no. 1999, p. 688, 1977.

[19] C. Hill, R. Bellamy, T. Erickson, and M. Burnett, "Trials and tribulations of developers of intelligent systems: A field study," in *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*, 2016, pp. 162–170.

[20] Y. S. Yoon and B. A. Myers, "A longitudinal study of programmers' backtracking," in *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, 2014, pp. 101–108.

[21] M. P. Robillard, "What makes APIs hard to learn? Answers from developers," *IEEE Softw.*, vol. 26, no. 6, 2009.

[22] D. W. Sandberg, "Smalltalk and exploratory programming," *SIGPLAN Not.*, vol. 23, no. 10, pp. 85–92, 1988.

[23] E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1–11.

[24] Z. Merali, "Error: Why scientific programming does not compute," *Nature*, vol. 467, no. 7317, pp. 775–777, 2010.

[25] "scikit-image: image processing in Python." [Online]. Available: http://scikit-image.org/.

[26] S. K. Kuttal, A. Sarma, and G. Rothermel, "History repeats itself more easily when you log it: Versioning for mashups," *Proc. - 2011 IEEE Symp. Vis. Lang. Hum. Centric Comput. VL/HCC 2011*, pp. 69–72, 2011.

[27] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012, pp. 2951–2959.

[28] M. Terry, E. D. Mynatt, K. Nakakoji, and Y. Yamamoto, "Variation in element and action: supporting simultaneous development of alternative solutions," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004, pp. 711–718.

[29] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey, "Software history under the lens: a study on why and how developers examine it," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, 2015, pp. 1–10.

[30] S. B. Davidson and J. Freire, "Provenance and scientific workflows: challenges and opportunities," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1345–1350.

[31] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Evaluation and Usability of Programming Languages and Tools*, 2010, p. 8.