Human Factors Affecting Dependability in End-User Programming

Andrew J. Ko and Brad A. Myers
Human-Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
ajko@cmu.edu, bam@cs.cmu.edu

ABSTRACT

Human factors affecting the dependability of end user's programs are discussed in the context of controlled and observational studies of both professional and end-user programmers. These factors include the influence of the types of behaviors that end users wish to implement, end user's fundamental cognitive biases, barriers in the languages, environments, libraries, and other tools used by end users, and end users' difficulties with understanding their code's meaning and execution.

Categories and Subject Descriptors
D.2.6 [**Programming Environments**]: Integrated environments.

General Terms
Design, Human Factors.

Keywords
End-user programming, human factors.

1. INTRODUCTION

The goal of project Marmalade (www.cs.cmu.edu/~NatProg) is to design innovative programming environments, tools, and interaction techniques that significantly lower the barriers to successful programming. An important part of achieving this goal has been to better understand the barriers in programming systems that make it difficult for both professional and end-user programmers to be successful. This has involved several empirical studies of programmers, both observational [3, 4] and controlled [5, 6], using various programming systems including Alice [2], Visual Basic.NET, and Macromedia Flash and Director.

In this paper we would like to share some of our more general insights from these studies in the hopes of fostering discussion about some of the central factors affecting the dependability of end users' programs. In summary, these factors include:

- What end users want their programs to do;
- Fundamental cognitive biases that can cause end users to introduce errors into code;
- The languages, environments, libraries, and tools used by end users to create their programs;
- The code that end users create; and,
- The errors in the code that end users create.

We will end our discussion with some insights on the implications for the design of end-user programming environments.

2. What End Users Want Programs to Do

One factor that influenced end-user programmers' success in our studies was the behaviors that they wanted their programs to perform.

In many cases, our participants found that the algorithm they desired was in many ways more complicated than the code required to implement it. For example, when participants were required to implement their own sorting algorithm for a list of names, it was the algorithm itself, and not the code that they had to write to implement the algorithm, that caused them the most difficulties and the most errors.

In other cases, end-user programmers' expected a particular behavior to be straightforward to implement, but found that several other things had to be implemented in order to achieve the behavior they desired. For example, in our study of Visual Basic.NET [7], we observed one student try to create an alarm clock that would play digital music. After a few hours of simply trying to get a timer to count seconds, he decided to abandon the digital music idea, and just focus on getting the alarm clock to work.

We have also observed that end-user programmers sometimes found that the behavior they desired was beyond the scope of the programming system's abstractions. When this occurred, they were forced to either solve a problem in a very cumbersome and unintuitive way, frequently leading to errors, or else find a different programming system that offered more suitable abstractions. For example, many of the participants in our study of Visual Basic.NET wanted to create animations, but found that its support for animations was minimal. Many programmers, rather than move to a programming system tailored towards animation such as Flash, instead tried to find workarounds for animation by searching on the Internet. Programmers reported that they had already invested so much into one programming system that learning another would not be worth their time.

3. End User’s Cognitive Biases

End-user programmers, like all people, have several fundamental cognitive biases that directly affect their ability to create correct programs. We identified and discussed these biases in detail in an article on the cognitive causes of software errors [6]. In summary, these biases follow a simple pattern: when given the choice, people tend to follow the path of least resistance. For example:

- We tend to collect only enough information needed to make *a* decision, and not necessarily the *best* one. Consequently, we frequently make misinformed decisions, simplifying assumptions, and false hypotheses. For example, in all of our studies of debugging, end-user programmers tested the first hypothesis that came to mind, not necessarily the best hypothesis, and certainly not all of the hypotheses.
- We tend to use the tools that let us reach our short-term goals the quickest, regardless of their impact on our long-term goals. For example, even when end users in our studies of Alice had the long-term goal of creating well-parameterized, extensible animations for use in many circumstances, they avoided parameterizing the animations entirely because it allowed them to implement the first animation more quickly. This, in turn, made the animation code more difficult to reuse for later projects.
- We tend to prefer more immediate, but less useful feedback over more delayed, but more useful feedback. For example, in our studies of debugging, programmers used print statements because they would quickly get some data about their program’s execution, even when using a breakpoint would have given them more accurate and concrete data about their particular debugging problem.
- We tend to prefer simple explanations for phenomena to more complex ones; in particular, we often believe that there is only a single cause behind some phenomena, when in fact there may be multiple. In our studies of debugging, users generally only considered one possible cause of a program failure at a time, even if there were in fact multiple. Furthermore, when one cause was repaired but the other causes still resulted in some failure, users assumed that the repair must not have been necessary and often undid it.
- We tend to believe that events that are correlated also have some causal relationship. For example, several times in our studies, users’ programs exhibited some failure shortly after they made some change, and the user believed that their recent change was the cause of the failure. It many cases, however, the recent change had nothing to do with the failure; it was actually due to some other error that was coincidentally manifested at the same time.
- Hypotheses that we form with impoverished data tend to interfere with our interpretation of new and more accurate data, leading to oversimplified or faulty models of a problem space. For example, when users in our studies copied and pasted code, they often tested it with a single test case, and later, when seeing their program fail, overlooked the copied code as a potential cause of the failure due to the earlier assumption of its correctness.

In our studies, the effects of these cognitive biases were not limited to any particular part of programming activity: we have seen them cause problems when end-user programmers are writing, changing, testing, understanding and debugging code.

4. Languages, Libraries, and Tools

Another factor that affected end-user programmers’ success in our studies were the programming languages, environments, libraries, debuggers, and other tools used by end users. When we studied Visual Basic and Macromedia Flash [7], we found that each part of a programming system has a user interface like any other software tool—even the languages and libraries—and that each one posed specific barriers to end users’ success:

- Language syntax was a significant problem, despite each environment’s attempt to offer support for repairing syntax errors. This was largely because users did not know the syntax or how to learn it. Many of the participants in our study of Visual Basic.NET admitted that, were they not required to learn the language for a class, they would have stopped trying within the first week because of their trouble with syntax. This suggests that end-user programming systems need new, more learnable interaction techniques for constructing code. We are currently working on this problem, designing new approaches to structured editing [8].
- If users were comfortable with a language construct, method call or other tool, they often tried to use it in inappropriate ways when they perceived a high cost in finding and learning to use a new and more appropriate tool, or when they did not know such a tool existed. For example, many users in our study of Visual Basic.NET became accustomed to using *for* loops and avoided learning how to use other loops, even when they had trouble using the *for* loop for a particular task.
- Oftentimes, the sheer number of ways to implement a behavior in Visual Basic and Flash was a problem. For example, when using Visual Basic, users found two ways to obtain the current date, three ways in which the dates and times could be stored, and nearly a dozen ways to keep time. Consequently, *choosing* an approach to implementing a behavior was often more difficult than implementing any one of the approaches, because they did not know which would actually suit their needs.
- In many cases, users could only accomplish a task through the coordinated use of two or more language constructs, API calls, or other tools, but figuring out how to use the them together—or how *not* to use them together—was never straightforward. For example, nearly all of the students in our study of Visual Basic spent hours determining how to pass data from one Visual Basic form to another.

In all of our studies, a common way that end-user programmers overcame these barriers was through informal apprenticeships: less experienced programmers consulted with more experienced programmers in order to solve or better understand a problem. One idea is that end-user programming systems could offer ways of helping less experienced users find more expert users [11].

Another way that end users overcame these barriers was to find example code on the Internet and adapt it for their purposes. While this frequently helped them make progress, it almost always led to the introduction of errors. The example code often contained errors itself, or adapting the code was not straightforward because important context was missing. We are interested in investigating ways that end-user programming systems could help find example code based on the type of behavior that users want to implement, and provide support for integrating the example into their code.

5. The Code

One thing that makes programming unique is that it involves the creation of an artifact that will be interpreted by a machine [1]. Consequently, end-user programmers, as with anybody who programs, must have some sense of how this machine will interpret what they have created. In our studies, however, after end-user programmers created code, they often did not know what it meant or how it worked, let alone how a computer might interpret it; this was often because they had only succeeded with the help of others, through face-to-face help or example code. Many participants said, "I don't know why this works, but I'm not going to change it..." or "I don't remember how I did this, and I'm not eager to find out." When asked to describe a particularly complicated block of code, one said, "Oh, that's some magic I found on the web. It does what I need it to, but I have no idea how." Because of this lack of understanding of their own code, end users frequently introduced errors when they had to modify it.

When end users executed their code, their lack of understanding about how the computer interpreted their code led directly to difficulties understanding why their program behaved as it did. In all of our studies, when end-user programmers observed their program fail, they always reacted with a "Why did..." or "Why didn't..." question about their program's behavior. For example, in our studies of Alice [5], they asked, "Why didn't Pac-Man resize?" or "Why didn't the big dot disappear?" The program's output, often the most familiar part of the program to the end-user programmer, was the most salient thing to ask about, but also the most difficult thing to answer. Users had to:

- Think of a question to ask;
- Think of a possible answer;
- Think of a way to verify their hypothesis; and
- Think of an alternative explanation after finding out the first was wrong.

Not only were each of these steps prone to the cognitive biases discussed in Section 3 (such as choosing false hypotheses based on a limited understanding of how the machine interpreted their program), but also the programming environments provided no support for accomplishing these steps. Furthermore, when thinking of a way to verify their hypothesis, most end users chose to modify their code in some way instead of collecting data to test their hypothesis.

End-user programmers would benefit from tools to help consider various hypotheses, helping to remove any bias toward any one particular hypothesis, as well as tools to help test a hypothesis by collecting information related to the hypothesis. Our Whyline debugging tool addresses all of these problems [5].

6. The Errors

In our studies, we found that all of the factors discussed thus far—the programs that end-user programmers want to write, their inherent cognitive biases, the tools they use, and the code that they create—were in some way responsible for the introduction of errors in their code. But in many cases, errors were also indirectly responsible for further errors. For example,

- When trying to debug one error, many users mistakenly attributed the cause of a failure to a correct fragment of code, only to modify the correct code in an attempt to repair the error, introducing new errors.
- Many users, after long periods of fruitless debugging, decided to delete all of the code that they thought was erroneous, and start over. This was particularly problematic when the code that they deleted was not broken, since none of the end users kept version histories of their code.
- End users frequently introduced errors because of some false assumption, and after testing their program and believing it had succeeded, also believed that their false assumption was confirmed, leading to further errors due to the same assumption. For example, when using Flash, end users frequently had animations that looked quite similar. When they created code to go from one animation to another, but using incorrect parameters, when testing, they often believed that the code worked because the animation looked similar. They then continued to use the incorrect parameters in other code.

These situations, being quite common, suggest that if end-user programming systems can prevent a single type of error, they may actually be preventing a whole class of potential errors. Further research is necessary to determine what types these might be.

7. What To Do?

All of our studies' findings, combined with the decades of research on the psychology of programming [9], suggest that programming requires an acute attention to detail—something which is in direct opposition to decades of research on human error [10] that suggests that people are optimized for making decisions that are merely "good enough" for the current situation.

As programming system designers, what can we do about this? We certainly cannot change human nature. While software engineers are trained to suppress their human nature by being thorough, planning ahead, and using process and methodology to their advantage, we can make no such assumptions about end user programmers. Most end users will learn just enough about a tool to support their primary task, and would not even think to use a process—they have their own processes in their primary work activities to worry about.

We can, however, change the programming systems that end users interact with. To start, we can design programming systems that help end-user programmers attend to "important" details. Otherwise, they will be solely responsible for deciding what is important to attend to, and we know from extensive research on human error that people make biased, short-term assessments of importance. We can also minimize the time that end users have to spend on "unimportant" details by having the programming system do any work that the programmer need not be involved in. For example, if at some point the programmer will have to find all of the valid method calls for an object, have the computer do the searching for them, since it is much more objective and thorough than the end user.

The next obvious question is, what are the "important" details? In some sense, only the end-user programmer knows what is important, since they are the only ones who understand what they want their program to do. As programming system designers, then, one way to assess the "importance" of some detail is to determine the degree to which it minimizes the influence of end-user programmers' own biases on their decisions. By minimizing this influence, we may maximize end-user programmers' ability to achieve the goals they *intend* to achieve, were it not for their inherent subjectivity.

Under this definition, we can make several design suggestions:

- Instead of having users generate their own hypotheses about the cause of a runtime failure, have the programming system provide a more objective and exhaustive list of possible explanations and have end users choose from them.
- Instead of having users collect information about their program's runtime execution manually via print statements and other facilities, have the system collect it for them, and then allow them to evaluate it relative to the behavior they expected.
- Instead of having users guess what values a variable had during the last execution of the program, show them a complete list of the values so that they can verify them relative to what they expected.
- Instead of having users conceive of their own design patterns for using an API, give them reusable templates that have been thoroughly tested and carefully designed to support common tasks.
- Instead of expecting users to *recall* a language syntax, design interaction techniques for editing code that allow them to simply *recognize* the syntax. This might involve the drag and drop interactions in Alice [2], or new types of structured keyboard-based interactions that mimic interactions with freeform text [8]. This would also free users from having to manage the layout of text in order to keep it readable.
- Instead of expecting users to remember their remaining development tasks, remember their tasks for them by supporting to-do lists that are both embedded in context and aggregated globally in the environment. Better yet, programming systems could generate to-do list items *automatically* by, for example, identifying unhandled cases in a set of conditionals, noting procedures that have yet to be called, and finding variables that were assigned some value that was never used.
- Instead of requiring users to manage copies of code manually, offer facilities that identify copied code automatically and either help users generalize their copied code, or simply maintain the relationships between the original and copied code. In the latter case, when the original code changes, users could be reminded and asked what action to take, if any.

The common theme underlying all of these examples is that both parties in the interaction do what they do best: programming systems are responsible for being objective, deterministic, and thorough, and end-user are responsible for being creative and judging whether program's behavior is what they expect.

8. Conclusion

We have summarized a number of human factors issues that affect the dependability of end user's programs, based on several observational and controlled studies of both professional and end-user programmers. We are currently working on several new tools, based on our findings:

- The Whyline [5], a debugging tool that lets end users ask questions about their program's failures in terms of its output and behavior.
- New structured editing interaction techniques that avoid the major usability problems with previous structured editors [8].
- A new toolkit for creating end-user programming systems that dramatically reduces the amount of work required to implement new tools and languages.

9. ACKNOWLEDGMENTS

This work was funded in part by the National Science Foundation, under NSF grant IIS-0329090, and as part of the EUSES consortium (End Users Shaping Effective Software) under NSF grant ITR CCR-0324770. The first author is also supported under a National Defense Science and Engineering Graduate Fellowship. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

10. REFERENCES

[1] Blackwell, A., First Steps in Programming: A Rationale for Attention Investment Models, *IEEE Symposia on Human-Centric Computing Languages and Environments*, Arlington, VA, 2-10, 2002.

[2] Dann, W., Cooper, S., and Pausch, R., Learning to Program with Alice: Prentice-Hall, 2003.

[3] Ko, A. J., A Contextual Inquiry of Expert Programmers in an Event-Based Programming Environment, *Human Factors in Computing Systems*, Fort Lauderdale, FL, 1036-1037, 2003.

[4] Ko, A. J. and Myers, B. A., Development and Evaluation of a Model of Programming Errors, *IEEE Symposia on Human-Centric Computing Languages and Environments*, Auckland, New Zealand, 7-14, 2003.

[5] Ko, A. J. and Myers, B. A., Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior, *CHI 2004*, Vienna, Austria, 151-158, 2004.

[6] Ko, A. J. and Myers, B. A., A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems, *To appear in the Journal of Visual Languages and Computing*, 2004.

[7] Ko, A. J., Myers, B. A., and Aung, H., Six Learning Barriers in End-User Programming Systems, *IEEE Symposium on Visual Languages and Human-Centric Computing*, Rome, Italy, 199-206, 2004.

[8] Ko, A. J., Aung, H., and Myers, B. A., Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing, *CHI '05: Human Factors in Computing*, Portland, OR, USA, (to appear), 2005.

[9] Pane, J. F. and Myers, B. A., "Usability Issues in the Design of Novice Programming Systems," Carnegie Mellon University, Pittsburgh, PA, School of Computer Science Technical Report CMU-CS-96-132, August 1996.

[10] Reason, J., Human Error. Cambridge, England: Cambridge University Press, 1990.

[11] Vivacqua, A. and Lieberman, H., Agents to Assist in Finding Help, *Conference on Human Factors in Computing*, 65-72, 2000.