Debugging Event-Driven Programming

James Davis
Virginia Tech
Blacksburg, VA, USA
davisjam@vt.edu

ABSTRACT
The importance of the event-driven programming (EDP) paradigm is on the rise. While EDP has been common in client-side applications since around 2000 (e.g. GUI development, JavaScript-driven web pages), it has only recently entered the mainstream in the settings of mobile computing (Android, 2008 [15]) and server-side programming (Node.js, 2009 [18]). The EDP paradigm is also a natural way to structure applications written for the burgeoning Internet of Things. However, little effort has been invested in understanding the processes programmers use to develop and debug EDP applications or in developing effective educational approaches for EDP. This research proposes to “debug event-driven programming,” exploring the mental processes EDP developers use to grok their code and considering contextualized educational approaches based on the students’ backgrounds. As a result, educators and systems researchers will be able to make informed decisions about the design of curricula, languages, development environments, and debugging aids.

A brief introduction to event-driven programming
Many universities dedicate much of their computer science curricula to procedural (imperative) programming, a sequential or multi-threaded programming paradigm in which programs follow a procedure (sequence of steps, “recipe”) in a relatively linear fashion until the completion of their algorithm [60]. The EDP paradigm [26] differs significantly, being non-sequential (asynchronous) and relational (webs of event handlers). I anticipate that these differences make EDP different to teach, learn, and practice than procedural programming.

The EDP paradigm is coming into its own. Though historically EDP has played second fiddle to procedural programming, the explosion of the event-driven Node.js server-side JavaScript framework [18] has demonstrated its relevance to applications running on the modern Internet. In an EDP system, the system defines the set of possible events (e.g. “mouse click” or “incoming network traffic on port 1234”), and the developer supplies a set of “event handlers” (also known as “callbacks”) to be executed when particular events occur [60, 24]. During execution in an EDP application, events are generated and then routed to the appropriate handlers.

EDP has been used at the level of hardware interrupts since the early days of computing, but to the best of our knowledge it first saw high-level use in the Mesa programming environment. Mesa was an early GUI system, and application developers could register event handlers for events like the re-sizing of a window [63]. EDP still holds sway in GUI environments, for example in the obsolete Java/AWT GUI framework [69] and the now popular Java/Swing and JavaFx GUI frameworks [20, 59]. EDP was later applied in early web servers, as researchers realized that it was particularly well suited to I/O-bound work like that of an FTP server. In this setting, network traffic is translated into client requests, which are “emitted” as events to be handled. For an FTP server, client requests are for files stored in the servers file system, so the handler would submit request(s) to the file system, registering yet another event handler to be executed once the file system completed the
request. Until the file system finished its request, the server could process other requests in a similar fashion. In this way, a single-threaded event server can handle multiple clients at once, with minimal overhead and high performance under certain workloads [52].

WHAT WE KNOW ABOUT PROGRAMMERS

There is a rich literature on programmers: the kinds of bugs they produce, how they debug, and the differences between novice and expert programmers. Much attention has been paid to the kinds of bugs produced by novices (e.g. [56]), as these bug classifications can be used as a feedback mechanism to refine the content presented in a course. Other work has gone into debugging techniques, examining debugging both as a specific form of troubleshooting [40] and as a skill to be taught in its own right [5].

Expert programmers

Research into how programmers debug began in the 1970s, beginning a long tradition of work on expert programmers. Gould and Drongowski initiated the field, assessing expert programmers in industry [28, 29]. Their employer, IBM, was understandably interested in understanding how its professional programmers did their work. They “seeded” one-page FORTRAN programs with errors and studied 30 and 10 experts as they debugged them. Using a combination of notes and think-aloud, they found that certain types of bugs were harder to identify than others.

Ten years later, Vessey compared the debugging processes of “novice” and “expert” professional programmers, using multi-page COBOL programs seeded with errors. Vessey found that experts outperform novices, noting that experts seemed to be better at “chunking” a program: experts could divide a program into its related components more readily than novices, allowing them to focus on the chunks relevant to the bugs [65]. We note that these “novices” are professional programmers, not to be confused with the research on true novices in the university setting. This research is discussed below.

Eisenstadt provided a qualitative description of debugging in industry [22], collecting “debugging war stories” from 78 professionals. He reported commonalities in the stories: many of these experts employed data gathering approaches (e.g. printf) and hand simulation to better understand the program and the cause of the bug, and half of the bugs were cases in which the professional tools (e.g. debuggers) available to the developers were useless due to significant spatial or temporal gaps between a bug’s root cause and its symptoms.

Inspired by Eisenstadt’s discussion of the importance of data gathering, and the use of information foraging in earlier work by Russell et al. [58], Lawrance et al. studied the way experts interacted with a large and unfamiliar software project from an information foraging theory perspective [42]; they found that programmer behavior could be modeled using information foraging without needing a cognitive model. This model was also confirmed in a study of end-user programmers as they evaluated a large, unfamiliar spreadsheet [30].

Novice programmers

Research on professional programming practice has been paralleled by research on the practices of true novice programmers. This line of research considers novice programmers, which are students in their first or second computer science course (CS1 or CS2). Research in this area studies the ways novice programmers interact with source code and programming aids while programming and debugging, typically relying on deep investigations of a small group of students. Though the programming languages have changed from COBOL and BASIC to Java, the findings of the earlier generations of researchers have been replicated by more recent work.

The study of the bugs created by novice programmers was spearheaded by Spohrer and Soloway, who argued that an empirical understanding of novice bugs would lead to improved courses. They structured their paper around two pieces of folk wisdom: that most errors are due to a small number of root causes, and that most errors are due to misunderstandings of the programming language. Their data agree that a few root causes explain most errors, but basing their work in GOMS philosophy [38] they argue that programmer errors are due to the programmer’s incorrect goals and methods unrelated to language misunderstandings [61].

The oft-cited work of Katz and Anderson discussed four experiments with between 10 and 40 participants enrolled in a CS2 course. The most common bug location strategies these students used were “simple recognition”, “forward-looking” approaches like hand simulation, and “backward-looking” approaches like causal reasoning (reason backward from output). Of particular interest were the findings of their third experiment, which showed that students take different bug location strategies depending on whether or not they are familiar with the code. Students took forward-looking approaches when working with unfamiliar code, preferring backward-looking approaches for code they understood [40].

Perkins and Martin studied 20 high school students in their second semester of a programming course [55]. By monitoring each student’s performance as they completed a set of asterisk-printing problems, they found that these novice programmers often ran into problems attributed to their “fragile knowledge”. Correct programming relies on the successful integration of a rich set of concepts, and partial, hard to access, or mis-applied knowledge often tripped up these novices.

Ahmadzadeh et al. innovated a two-pronged approach to studying CS1 students: they assessed both programming practice over the course of a semester and debugging skills in a 2-hour exam. From this rich dataset they showed that good programmers (as measured by course grades) are not always good debuggers! This finding shows that students don’t always learn debugging by themselves by working through homework assignments, and that explicitly teaching debugging might be necessary [5]. Murphy et al. agreed with this finding in their study of 21 students enrolled in CS2 across 7 universities: their students had difficulties applying debugging techniques consistently and effectively [46], smacking of the fragile state of novice knowledge identified by Perkins and Martin [55].
Though many authors characterize the bugs introduced by their students on a small scale, two particularly significant efforts have been made to identify the issues encountered by students in CS1. The first was a “critical incidents” style approach that surveyed instructors at 50 colleges for the most common kinds of bugs introduced by CS1 students [33], while the second painstakingly recorded each interaction between roughly 400 students and their teaching assistants at the University of Otago over 2 years. In doing so, Robins et al. were able to identify (year 1) and corroborate (year 2) the most common types of bugs introduced by students in CS1 [56]. Hristova et al. classified programming errors as syntax, semantic, or logic, while Robins et al. favored the higher-level labels of background, general and specific to account for the more open-ended questions asked of the teaching assistants. Hristova et al.’s error categories fall into Robins et al.’s “general” and “specific” categories, while Robins et al.’s approach identified the additional category of “background” issues like problems with the operating system or understanding the assignment. While working in pairs, Hanks’s students replicated the problem distribution described by Robins et al., although each pair experienced fewer issues overall [31].

Other lines of research on programmer behavior

Other avenues of research have pursued predictors of student success in computer science coursework. Researchers have investigated the relationship between programming ability and a variety of psychological measures of programmers. The most popular psychological measures to attempt are field dependence/field independence [67] and the Myers-Briggs personality types (MBTI) [48]. Though we presume the reader’s familiarity with the MBTI, field dependence and field independence describe how readily a person can perceive a complex object as composed of individual parts; field dependence implies difficulty in breaking down the whole into its parts, or to perceive an object outside of the context in which it is observed. While much of the research looks for relationships between psychological measure and course outcome (11, 17, 47, 68), a more promising approach seems to be to assess small aspects of the process: individual assignments [11] or even the individual stages of problem-solving in the context of programming [10]. The effort to correlate programming ability and psychological measures has been largely unsuccessful, as the most consistent finding has been that academic success begets academic success (e.g. [47]). However, some promising research demonstrated that field independent programmers are better at algorithmic design [10] and locating logical errors [14].

Researchers have considered still other aspects of programmers, including verbal ability and gender. For example, in a study of novice debugging strategies, Romero et al. found no relationship between verbal ability (as measured by GRE test questions) and debugging skill [57]. Others have investigated gender, e.g. finding in end-user programmers that females introduce more bugs than males while debugging a program seeded with errors [8], and that males tinker more than females during end-user debugging, sometimes to their detriment [9]. From another angle, Woszczynski et al. observed no statistical difference in the performance of males and females measured by final grade in a CS1 course [68].

WHAT WE DON’T KNOW ABOUT PROGRAMMERS

Despite decades of research into the practice of programming by experts and novices, many questions remain unanswered. We observe two fundamental gaps in the literature: the effect of programming paradigm and the effect of programmer background.

Programming paradigms

First, almost all of the research on novices discussed in Section has focused on the procedural programming paradigm, in which students give the computer a sequence of steps to accomplish the desired task. This decision is presumably an artifact of history: these are the languages used in the CS1 and CS2 courses taught by the researchers, and therefore are the easiest for the researchers to study. There is no reason to suspect, however, that the conclusions of these researchers about issues encountered by novices can be generalized to other programming paradigms. Indeed each paradigm is different, with its own strengths and weaknesses [64]. Exceptions to the “procedural club” are LISP [40] (functional paradigm, though paradigm was not emphasized in this study) and Excel (logic paradigm) [30]. The rest use strictly procedural languages or are heavily reliant on the procedural components of languages: Pascal [61], BASIC [55], and Java [56, 57, 31, 33, 46, 45]. What impact might the use of different programming paradigms have on novice performance?

We were able to locate only two papers that explicitly consider the impact of different paradigms in some way. Myers et al. compared cognitive style to ability in programming paradigm based on non-novice students taking two post-CS2 courses concurrently, with inconclusive results [47], while White and Sivitanides proposed a theory predicting performance in languages falling into different paradigms as a function of cognitive development and brain hemisphere preference, but made no attempt to validate their theory [66]. We propose to enhance the research methodologies in the field of novice programming errors with techniques to give us a deeper sense of the students’ cognitive processes, and to apply these methodologies in the setting of disparate programming paradigms.

Programmer background

Second, the research on programming errors in Sections and assumes the programmers are homogeneous in every respect other than amount of programming experience. Though some list the genders or ages of the participants, none of the analyses distinguish between participants on the basis of these differences. This is a curious state of affairs. We suggest that this lack of investigation is due to the relatively small sample sizes of many of the studies, perhaps due in turn to the man-hours required to follow the verbal protocol methodologies on which they rely. Does a programmer’s background have an effect on his ability or on the kinds of problems he experiences?

One aspect of programmer background of particular import in the modern era is cultural background. In addition to mechanical labor like manufacturing, Western companies large
and small have outsourced intellectual work like computer programming. We have observed this phenomenon first-hand while working at IBM: American employees were occasionally replaced by Indian or Chinese employees. Understanding the ramifications of changing the cultural background of their programmers could help companies determine which projects are better suited to outsourcing. To understand the issues in this area, we draw on the pioneering work of Nisbett et al. and Nisbett summarized in [50, 49].

Cultural background
Nisbett et al.’s argument is complex, drawing relationships between societal structure and cognitive processes, and we summarize it here. First, the social organization of ancient Greek and Chinese cultures was markedly different. Greek culture was focused on the individual, while Chinese culture was focused on the group. This difference affected their perspective on how the natural world worked (indeed, the Chinese barely acknowledged the concept of “nature as distinct from human or spiritual entities”). Due to their focus on the individual, the Greeks were prone to systematizing everything, looking for rules and categorizations in the nature. In contrast, the Chinese did not construct formal models of nature. Nisbett et al. argue that the social elements of life in these ancient cultures affected their systems of thought, in turn influencing their cognitive processes. Of particular relevance to our proposed research, the ancient Greeks became in general more oriented to individual parts, while the Chinese considered parts inseparably from the whole.

While descriptive of life in ancient Greece and China, are these differences relevant today? Nisbett et al. cite anthropological evidence that the social order of ancient Greece is similar to the social order of modern North America and Europe (Westerners), while that of ancient China is reflected in the social order of Eastern Asian countries (China, Japan, South Korea, etc.) (Easterners). Nisbett et al. contend that “the original differences in cognitive orientations were due to the social psychological ones”, and cite a broad array of evidence demonstrating these cognitive differences between Westerners and Easterners in the modern age. We are principally concerned with the ancient distinction between the individual (Greece) and the whole (China) and its implications for computer programming. Whether a person is more attuned to individual components or to the whole can be measured using Witkin’s concept of field dependence/independence (FD/FI) [67]: Westerners are more field independent, identifying individual objects as they compose a more complex whole, while Easterners are more field dependent, considering the whole composed of inseparable parts [50].

We propose to apply Nisbett’s research on the cultural basis of cognitive processes to the area of computer program debugging, which will help educators and employers understand the potential weaknesses of their students and employees based on their cultural background. Despite over 3000 citations of the work of Nisbett et al. [50], it has made few inroads into computer science research. Though Pauleen et al. appealed for this research in 2005 [54], the computer science education community does not seem to have heeded their call. We will.

RESEARCH QUESTIONS
1. **RQ1** Do the problems encountered by novice programmers vary across different programming paradigms? Are the problems and their distributions a function of the programming paradigm?

2. **RQ2** Does a student’s background affect the kinds of issues they encounter in different programming paradigms?

To answer these research questions, we design our experiments to investigate the following hypotheses:

1. **H1** Novice programmers will experience different kinds of issues when being taught using different programming paradigms. For issues common to all paradigms, we do not expect the distribution of issues to vary significantly by paradigm. This hypothesis will be tested by comparing the event-driven paradigm to procedural programming.

2. **H2** Culturally Western programmers and culturally Eastern programmers will differ in their ability to perform different kinds of activities in event-driven programming.

Our first hypothesis is open-ended, mirroring the fact that the literature on novice errors is exclusively focused on the procedural programming paradigm. This aspect of the study will be exploratory because the literature contains no hints as to the types of errors expected in other paradigms.

We base the cultural aspect of **H2** on the differences between Easterners and Westerners described by Nisbett et al. [50] and Nisbett [49], discussed earlier in more detail (Section ). We predict, in line with previous research [14, 10] that Westerners will be more effective at the intra-function aspects of EDP, while Easterners will be more effective at dealing with the inter-function relationships necessary in EDP.

What is a novice programmer?
We will study novice programmers in answering both of these research questions. It is difficult to consistently define a “novice programmer” across different institutions and different curricula. For example, the novices of Katz and Anderson were described as only making apparently-accidental “slips”, while students at other universities at a similar point in their academic careers are described as struggling with much more basic concepts at a similar point in their academic careers [33, 25, 31]. Many factors can influence student ability, including the skill of the educator, the caliber of the students, and any prior programming experience they might have. A strong experimental design must control for these factors to obtain a sound definition of a novice programmer. For ease of comparison, we will study only students at Virginia Tech who have taken the same set of computer science courses.

**RESEARCH QUESTION 1**
We wish to understand whether novices experience different issues when working in different programming paradigms. We hypothesize that they will indeed do so, and will establish this by comparing the issues they encounter when being trained in event-driven programming compared to procedural programming. This research question relies on concepts from HCI paradigms I and II [32].
Experimental design

Studies characterizing student issues are typically run for the entire duration of a programming course. We will therefore study students taking a programming course emphasizing the procedural paradigm and students taking a programming course emphasizing the event-driven paradigm. The Java programming language can be used for either paradigm, by emphasizing the existing introductory material (procedural, with light training in object-oriented programming) or by orienting the course towards GUI programming. We propose focusing Virginia Tech’s Java-based CS1054 course [2] (CS1) in these two directions in two consecutive semesters, comparing the errors experienced by these novices under each paradigm. The current course focuses on the procedural paradigm, but the same material can be presented in the event-driven paradigm as demonstrated by previous CS1 courses taught in the event-driven paradigm [13, 16]. As a programming-intensive course, CS1054 is more suitable for studying novice errors than other non-major courses like CS1014 [1].

Data collection protocols

In this section we discuss the methodological techniques used by researchers in this area. Though other techniques are mentioned in the literature, the primary approaches whose data is cited during the discussion of results are verbal protocols and on-line protocols. After discussing the use of these protocols, we propose to combine them in a novel way.

Verbal protocols

Researchers frequently use verbal protocols [23] to get inside programmers’ heads and understand their cognitive processes. Ericsson and Simon observe that verbal protocols won’t significantly interfere with participants’ cognitive processes provided that the participant encodes the information verbally, and that if the information is encoded in some other form (e.g. visual data, smells) the participant will need to devote mental resources to vocalize that information. Though some researchers uncritically use verbal protocols for tasks involving visual representations of data [57], the vast majority of uses seem in keeping with Ericsson and Simon’s guidelines.

On-line protocols

A promising, more data-oriented branch of research makes use of on-line protocols, introduced by Spohrer [61]. On-line protocols require the student to use the researcher’s development environment, tracking each student’s every move. Researchers typically capture coding snapshots of each programming assignment each time the student compiles their code, and researchers subsequently categorize the syntax errors made by novices [5, 36] or identify and explain semantic and logical errors based on psychological analysis of the source code [61]. On-line data can of course be used in tandem with qualitative data [19]. On-line protocols, also known as “educational data mining”, have increased in popularity in recent years thanks to a shift towards online programming environments that facilitate fine-grained data collection methods [34].

Our proposed protocols

On-line protocols A shortcoming of on-line protocols is that the data do not provide explanatory power. Researchers can determine whether code contains syntax errors or whether it fails a particular test case, but they cannot understand the programmer’s intent without pseudo-scientific psychological analysis; Spohrer and Soloway’s work acknowledged this concern, writing that “the[ir] plausible accounts [of programmer’s thought process]...must be subjected to further scientific inquiry” [61]. As far as we are aware, no subsequent work has identified a good quantitative scheme for determining the programmer’s thought process. We propose a two-pronged approach with a novel methodological twist that promises to give deeper insight into the programmer’s thought process.

We agree that on-line protocols promise the best source of scalable quantitative data about a programmer’s thought process. However, a novice’s source code alone – often lacking comments or good variable names – leaves much to be desired as a window into their mind. To gain this missing insight into a student’s thought process, we propose modifying the Java grammar to require a comment after every statement and preceding every block of code. This novel twist on on-line protocols will give us a sense of why each line of code exists.

Furthermore, we can combine this scheme with our own application test suite based on the specifications of each programming assignment. We agree with prior work [61, 36, 5] that compilation time is a good level of granularity. By filtering for programs that are syntactically correct (those that pass compilation) [61], we can determine the tests failed by each “valid” program, and use recent advances in program-level root cause analysis (e.g. [39]) to pinpoint the line(s) of code responsible for each test failure. Due to our changes to the Java grammar, the compiler will reject any programs without comments on every line; since these programs are syntactically correct, they will be highly commented. We can then analyze the comment on each line responsible for the test failure to understand the true root cause of the failure. We know that “the [root cause of the] error is between the chair and the computer”, but with the programmer’s comments in hand we can understand the thought process that led to the failure.

Virginia Tech already employs an online programming environment in the form of the Web-Computer Aided Testing (Web-CAT) environment, used e.g. in CS1114 (CS1 for majors) [3]. Since students do their programming within this environment, we have the opportunity to track their behavior at granularities ranging from key strokes to assignment submissions. We will modify the Java compiler in the Web-CAT environment to reject programs without comments as discussed above, and to record each compilation attempt for subsequent analysis.

Example 1: Procedural programming Figures 1 and 2 illustrate the potential explanatory power of our comment-every-line on-line protocol. Figure 1 is pseudocode for a function that doubles the value of each number in an array, returning an array of doubled numbers. It contains an error: on line 4, the programmer computed the doubled value, but fails to insert it into the doubleNums array defined on line 2. The result is that on line 5 the programmer always returns an empty array. We can only guess at the gap in the programmer’s line of reasoning – Did they simply forget to add it to doubleNums? Having
we recognize that not all comments are created alike, and feel

Verbal protocols with confidence as an S11 in the notation of Robins et al. [56]. However, we anticipate the need for a think-aloud verbal protocol. However, the commented on-line protocol has protected us from an incorrect analysis of the programmer.

Example 2: Event-driven programming No novice errors specifically associated with the event-driven programming paradigm have been identified. In Figures 3 and 4, we illustrate a novice error inspired by [12]. The assignment is intended to allow a user to draw lines on the screen. The expected behavior is as follows: when the user clicks (OnMouseDown) and releases (OnMouseUp), a line $L_1$ is drawn between the Down and Up points. When the user clicks again, a line $L_2$ should connect the endpoint of his previous line to the beginning of his new line, and the new line should be drawn as already described. However, the hypothetical novice code in Figure 3 will only draw the $L_1$-style between mouse clicks, and won’t draw any $L_2$-style lines connecting the $L_1$-style lines. From the uncommented code in Figure 3, it is not clear in what way the novice’s thoughts led to the error. At a high level, is the problem in comprehending the assignment, or in implementing it? Without comments or a conversation with the student, we cannot know.

Figure 4 illuminates the cause. Rather than an issue in implementing the assignment using EDP concepts, which might have incorrectly inspired an example of a new class of bug, the comment on line 1 provides evidence that the programmer simply misunderstood the assignment: they thought they were to draw lines only $L_1$-style lines between mouse clicks. We do not know whether the programmer would have been able to implement the assignment correctly had they understood it, but the commented on-line protocol has protected us from an incorrect analysis of the programmer.

Verbal protocols Nearly all of the studies cited earlier report the use of a think-aloud verbal protocol analysis. Our commented-on-line protocol should give us a “source code transcript” of each student’s thought process, so we do not anticipate the need for a think-aloud verbal protocol. However, we recognize that not all comments are created alike, and feel that verbal reports can provide some illumination. In addition to our commented on-line protocol, we propose the use of a retrospective verbal protocol along the lines of Flanagan’s critical incidents [27]. We will ask students for a description of how they fixed one or more of the bugs they introduced while working on each assignment. By administering these protocols as part of the assignment submission process, these reports should be fresh enough to provide helpful insights into the minds of novice programmers.

Data analysis

Data from our commented on-line protocol will give us a window into the programmer’s thought process. This will let us understand the kinds of errors to which novice programmers are prone, in both the procedural and the event-driven paradigms. By understanding these errors at a cognitive level, we can more carefully restructure the presentation of material to address common misconceptions or mistakes. We can apply transcription and coding techniques [23] to identify the most common types of errors. We anticipate that coding these source code transcripts will be far more feasible than attempting a think-aloud protocol, which requires students to program in a laboratory setting and researchers to carry out a painstaking transcription and coding of more meandering and gap-filled verbal transcripts.

Something that has troubled us about much of the research on the novice programming process has been the relatively small sample size: studies often use N=10-40 and are placed within a single university. Given the use of a labor-intensive think aloud protocol, we find these small sample sizes understandable. However, we are excited by the prospect of larger sample sizes afforded by the more compact and more readily-understood data offered by the commented on-line protocol we propose.

Data from our retrospective verbal protocol should further inform improvements in the presentation of course material. Interviews like those we describe have been administered in clinical settings (e.g. [28, 29, 25]), but we are not aware of studies using longitudinal per-student interviews [44]. We anticipate that the interview corpus will consist of a set of “Eureka!” moments from students, which can be used to help us understand effective debugging strategies and to give future students a sense of what the debugging process feels like from a qualitative perspective.

Threats

We have identified four threats to this study. The first is practical, while the remainder are threats to validity.

First, the CS department would need to be convinced to allow us to modify the content of one of their courses. We have proposed the use of the CS1054 course [2] because it is “CS1 for non-majors”. Modifying it will not affect the existing computer science curriculum for CS majors, so we conjecture that the CS department will be more likely to permit us to change its design. If necessary, we could scale down our method to introduce a 2-week module emphasizing either procedural or event-driven programming, the use of which might be more amenable to the course instructors.
Figure 3. Novice pseudo-code algorithm. Draws disconnected lines. What went wrong in the programmer’s thought process?

![Novice pseudo-code algorithm. Draws disconnected lines. What went wrong in the programmer’s thought process?](image)

Second, students could cheat our modification to the Java grammar by ending statements with empty comments or comments containing gibberish. We believe this tendency would be curbed by two factors. First, the instructor has both a carrot and a stick: he can emphasize the importance of comments as part of good programming practice, and he can punish this behavior by including the quality of comments in his grading rubric. Second, novices are not asked to write particularly long programs, so per-line comments should not be particularly arduous. However, our commented on-line protocol could be tiresome for students in higher-level courses (i.e. those above CS2). We suspect that modifying the Java grammar to only require comments at the block level would still make this protocol useful for higher-level courses.

Third, the instructors might not be able to comprehend the comments added by students. A novice might not be able to articulate why they are doing something. However, recall that Romero et al. reported no significant relationship between programming ability and verbal ability (as measured by GRE questions) [57]. If true, students of high verbal ability should be equally prone to writing code with and without errors, implying that at least some of the error-strewn programs will likely be authored by students with stronger writing abilities. These same students are expected to do a good job of keeping their comments in sync with their code. We anticipate, therefore, that at least a subset of the student programs will be comprehensible and suitable for study.

Lastly, it’s conceivable that being forced to justify every line of code with a comment will change the distribution of errors experienced by our participants. The distribution of errors in procedural programming has been reported and replicated [56, 31], and we will compare the error distribution between our procedural programming dataset and the expected distribution as reported by these other researchers. If it does not match, we will still be able to make a useful summary of error distributions, but we will know that the results will not be directly comparable to prior work. In the process, however, we will have inadvertently discovered a relatively easy way to help novice programmers write higher-quality code. We suspect many educators would appreciate this finding.

**RESEARCH QUESTION 2**

In the spirit of the Third Paradigm of HCI [32], we argue that not all programmers are created equal, and that the cultural context of a programmer may be particularly relevant to predicting their ability to debug different kinds of problems. We have predicted, based on the work of Nisbett et al. [50], that Westerners and Easterners will have different experiences while working on various aspects of EDP programming.

We expect the event-driven programming errors identified by the experiment discussed in Section to fall into two major categories: intra-function and inter-function. Event-driven programs have a procedural component, because they include a set of functions, each of which must work correctly. We refer to errors in this aspect of EDP programs as intra-function errors. Event-driven programs have a relational component as well: functions interact with one another as listeners for events, and must save state between events using techniques like closure. For example, in problems like that illustrated in Figure 3, a correct implementation requires the programmer to set the oldPoint variable to point in the OnMouseUp function for use in the OnMouseDown function. This principal is of course not unique to EDP, but in EDP this inter-function reasoning is a fundamental aspect rather than an advanced concept as in the procedural paradigm. Errors in this area will be referred to as inter-function errors because they require conceptualizing relationships between different invocations of the same or different functions.

Nisbett et al. [50] argued that Westerners are stronger at analytic thought and at considering each part of a whole in isolation, while Easterners excel at thinking of the role each piece plays in an indivisible whole. This difference manifests itself
in Witkin’s concept of field dependence/field independence (FD/FI) [67], with Westerners matching the description of field independence and Easterners that of field dependence.

We therefore refine our second hypothesis further: we predict that Westerners will be superior at designing and debugging intra-function algorithms, while Easterners will excel at designing and debugging inter-function relationships.

Experimental design
Our interest is in assessing the ability of Westerners and Easterners to design and debug particular aspects of problems in EDP programs. Though the techniques of our first experiment (Section ) will give us a sense of the experience of programmers in general when working in the EDP paradigm, the details of particular programming issues may be difficult to pick out. We therefore propose a clinical experiment in the vein of the studies of debugging methodology described earlier, patterned particularly after [10].

Measuring field dependence/independence
Table 1 shows the independent and dependent variables in this study. Though prior studies on the interaction of FD/FI and computer programming ability used the Group Embedded Figures Test to measure FD/FI [14, 10], performance on this instrument does not appear to be independent of culture. Bagley [7] observes that Easterners register as FI on EFT-style tests in contrast to their FD scores on other measures of FD/FI. He hypothesizes that this is because literacy in the languages of Eastern countries typically requires the memorization of and the ability to differentiate between hundreds or thousands of complex symbols. Nisbett et al. [50] refer to several other instruments suitable for measuring FD/FI, and following Ji and her colleagues we will use Witkin’s Rod-and-Frame Test (RFT) to measure FD/FI [37].

Data collection protocols
As is standard in this community, we will administer a 2-hour test to students in the EDP course discussed in Section . In the first 15 minutes, we will carry out an initial assessment of programming background and then measure the participant’s FD/FI status with the RFT instrument. We anticipate that students from an Eastern background (relatively common in programming classes) will score more strongly as FD, while students from a Western background will score as FI.

We will then administer a 20-minute warm-up task, asking students to debug two small (one-page) programs, one with an intra-function error and one with an inter-function error. Once the warm-up is concluded, the remainder of the time will be divided into three portions. First, participants will spend 45 minutes completing two design tasks, one calling for intra-function skill and the other requiring inter-function skill. We are not intimately familiar with the abilities of novice programmers or with appropriate expectations for their skills at EDP, so the particular tasks will need to be determined through an iterative pilot study. Second, participants will be asked to complete four debugging tasks in 45 minutes, each task featuring a different one-page program with one error. Half of the errors will be intra-function, and half will be inter-function. To avoid order effects, we will vary the order in which participants attempt the “first” and “second” tasks, as well as the order of the intra-function and inter-function problems within each task. The remaining time will be spent in a 15-minute post-test interview, using the retrospective protocol discussed in Section .

Data analysis
We will compare the field dependence/independence scores of the participants to their intra-function and inter-function debugging ability, measured by the speed and accuracy with which they identify the bugs during the six debugging tasks. We anticipate that introducing cultural background, measured by field dependence/field independence, will lead to the first (to our knowledge) successful demonstration of the strong prediction of debugging ability as a function of a psychological measure. The degree of field dependence/field independence variation across cultures is greater than within a culture [37], so the differences between Easterners and Westerners should be more pronounced than those within just the American culture.

It is our hope, therefore, that this experiment will demonstrate that instructors must consider the cultural context of each of their students. Computer science educators do not appear to have considered this possibility, possibly leading to poor learning outcomes for students from cultural backgrounds different from their instructor. Though we have deliberately selected the programming paradigm in which these differences seem the most likely to manifest, the notion of intra-function vs. inter-function errors is relevant in more complex programs, particularly in the popular object-oriented paradigm used in many CS programs. Software companies could also use the results of this study to assign employees to culturally appropriate tasks, for example asking Western programmers to work on procedural aspects of projects (e.g. business logic) and Eastern programmers to work on event-driven aspects of projects (e.g. the GUI).

Threats
First, it’s conceivable that not enough Easterners would enroll in CS1054 to provide a useful sample size. An alternative course in which this study would be a hypothetical graduate-level course that uses concepts from event-driven programming, e.g. a course in the IoT. From personal experience, I know that the graduate population of the “Systems” arm of Virginia Tech’s Computer Science department is largely Eastern, and these students would be interested in an IoT course. Since event-driven programming is an unusual paradigm, there should not be prior learning effects on the intra-function and inter-function debugging ability of many of these students. Furthermore, White and Sivitanides warned that not all first-year college students (nor indeed all adults) have attained full formal operational thinking, and as a result using a graduate-level course might actually be preferable for this study [66] if the results are to be generalized to professional practitioners.

Second, it’s possible that our predictions about the implications of field dependence and field independence are only relevant to larger programming projects. It is clear that both Westerners and Easterners can be skillful computer programmers, presumably because regardless of the programming
paradigm under consideration, both field dependent and field independent programmers can understand complex software system. A field dependent programmer would presumably understand a complex software system holistically, seeing the relationships between all of the components indivisibly, while a field independent programmer might understand it as the sum of the interactions between its constituent parts. The effect of field dependence or independence might not manifest at the scale of a one-page program, and a study like that of Lawrance et al. [42], observing professional programmers as they examine a large open-source code base, might be needed to truly observe any differences between individuals with these opposing traits. However, the relative ease of studying students in the university setting compared to professional programmers means that a “students first” approach is desirable.

CONCLUSION
EDP emerged as an important paradigm in the CS curriculum in the 1990s and early 2000s as GUIs became commonplace. Early explorations of EDP coursework focused on GUIs [62, 13, 16, 12]. While Stein argued that EDP should be included as one of several programming paradigms [62], others [13, 16] argued that it could be used as the focal paradigm of a CS1 course. There was some debate as to whether EDP was suitable for “CS1”-style classes, or if the concepts involved were too difficult. Bruce et al. argued that EDP could be used in CS1 [13], and indeed that doing so simplified teaching several key concepts [12]. However, research on EDP pedagogy has largely fallen silent: EDP did not “catch on” as a core concept in CS education, remaining a specialized topic for advanced students.

Ten years later, the IoT is increasingly imminent, and educators have begun to explore IoT-themed classes for advanced undergraduates and graduate students. These courses are intended for lower-level [41] or higher-level undergraduates [21, 51] or graduate students [43, 51]. Interestingly, the literature on undergraduate IoT coursework does not discuss pedagogical issues related to EDP, suggesting a potential barrier to student success: inviting students trained in one paradigm (e.g. functional or object-oriented) into a new programming paradigms (EDP) raises the specter of negative knowledge transfer [55]. Students may be stuck in one mode of thinking, unable to adapt to the new concepts in another paradigm, and mis-applying concepts from the first programming paradigm in the second [55].

We strongly feel that understanding the issues related to the EDP paradigm will inform computer science educators. This particular paradigm may be one of the keys to bringing the promise of the Internet of Things to fruition, and as a result understanding how to teach it is increasingly imperative. We believe the proposed research will inform educators as to the difficulties novices encounter in EDP, and that more broadly it will remind educators of the effect that cultural differences may have on their students’ abilities. These results will be useful not only for training students in the IoT, but also for other EDP settings like mobile development and GUI programming.

ACKNOWLEDGMENTS
We thank Professor Steve Harrison for his guidance on this project. We thank our advisor, Dr. Dongyoon Lee, for his tolerance of our interest in Human-Computer Interaction and Computer Science Education.

REFERENCES

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field dependence or independence</td>
<td>Independent</td>
<td>Measures how difficulty the participant can perceive field independent or context</td>
<td>Positive correlation to FD</td>
</tr>
<tr>
<td>Inter-function debugging ability</td>
<td>Independent</td>
<td>Measures how well the participant can debug interfuction errors</td>
<td>Positive correlation to FD</td>
</tr>
<tr>
<td>Intra-function debugging ability</td>
<td>Independent</td>
<td>Measures how well the participant can debug intra-function errors</td>
<td>Positive correlation to FD</td>
</tr>
</tbody>
</table>

Table 1. Independent and dependent variables in Experiment 2


