# Examining Active Error in Software Development

Tamara Lopez[1], Marian Petre[1] and Bashar Nuseibeh[1, 2]
[1]Centre for Research in Computing, The Open University, United Kingdom
[2]Lero The Irish Software Engineering Research Centre, Ireland
{t.lopez, m.petre, b.nuseibeh}@open.ac.uk

***Abstract*—Software rarely works as intended while it is being written. Things go wrong in the midst of everyday practice, and developers are commonly understood to form theories and strategies for dealing with them. Errors in this sense are not bugs left behind in software, they are actively encountered and experienced. This paper reports findings of an ethnographically-informed study undertaken to examine error encountered at the desk. Films depicting paired open-source development practice over the course of a month were analyzed to identify and delineate instances of active error. Instances were interpreted within a framework of error handling drawn from psychology research. Analyses of representative instances are given and discussed in relation to software engineering research that examines practice at the desk. Findings demonstrate that the significance of active error in software development is personal, shaped by passing time, the emergence of preferred practices and environmental changes.**

***Keywords—empirical studies; software development; software engineering; human error***

## I. Introduction

Error in software engineering is commonly described using terms like fault, defect or bug. These concepts represent elements of software in operation that threaten or produce undesirable or unintended deviations from specified behavior [1]. A bug is material, it can be tracked into source code, judgments can be made about what was done wrong and decisions taken about how it should be removed. It is not always possible to determine the circumstances under which a bug was written, or why [2]. Nevertheless, bugs are largely considered to be the result of human error and are often attributed to poor understanding, inexperience, lack of skill, or incompetence.

Using concepts developed in psychology and safety science, this paper considers error in a different sense. Errors are actively experienced, with effects that can be felt [3]. They arise at moments in which a planned sequence of mental or physical activities fails to achieve its intended outcome. Such errors are ephemeral, and as a result there are often few material traces [4] left within code, descriptions or project records.

In this report, the terms ***error*** or ***active error*** will be used to refer to errors that are experienced. ***Error handling*** will be used to describe the process by which errors are detected, identified and recovered from [3]. Instances of error handling will be referred to as encounters, incidents, problems or issues.

## II. Method

How do developers find and fix things that go wrong during software development? Error is commonly examined to assess why a software system suffered critical failure [5] or to improve software dependability [6]. Failures are considered to result from latent errors [3], and studies establish their causes by performing retrospective analyses on failed tests, bug or modification reports. Gaining access to software development sites is difficult [7], particularly when sought to examine mistakes [8].

Examination of active error requires naturalistic data [9]. Occurrences must be observable or reported by the people who experience them. With notable exceptions [10], error that arises during software development at points other than testing or integration is not well understood [8]. Problems in professional software development can take a while to solve [11]. Taken together, these points suggest a need for observation of practice over time, rather than by examining particular tasks or time-slices. Analysis must be prospective [12], following activity at the desk forward [13], rather than starting from outcomes and performing deductive analyses of events that occurred in the past [10].

The research reported here is one of three ethnographically-informed [14] studies in which the authors have examined error in software development practice. Research was conducted by observing developers in the field, and using data collected from interviews and gleaned through secondary observation [15] of films that depict software design and development activity at the desk. Data for this study was drawn from participant-created [16] video recordings of development and related sources including source code repositories, social media and blog posts. The materials were created in 2009, and are accessible on the Internet. Each film includes a screencast of desktop interaction and audio recording.

The pair of developers who created the videos granted permission for use in research. Pseudonyms have been provided to informants across studies. The names Marcus and Joe are used to refer to the pair in the text that follows.

### A. Sources

The films depict Marcus and Joe as they create extensions to an open-source, wiki-based acceptance test framework. They write stories within the test framework that define new functionality they intend to add, and use the Eclipse integrated development environment (IDE) to write classes and tests in the Java language.

ERC Advanced Grant 291652 - ASAP and SFI grant 13/RC/2094

978-1-5090-0252-8/16/$31.00 ©2016 IEEE

Marcus and Joe pair at the desk. One writes a unit test, defining proposed behavior for a class, while the other completes the implementation by adding classes or altering methods. The pair also use a Java interface written by Marcus some months prior to filming. This application programming interface (API) is referenced directly, and examples are consulted and borrowed from the documentation.

Each developer is familiar with the acceptance test framework, though Marcus has more recent experience in developing it. By contrast, Joe exhibits greater familiarity and responsibility for the development tools that are being used.

### A. Analysis

Principles of thematic analysis were used to segment, catalog and identify instances for examination. This method was selected because it is not overly structured, and analyses can be made independently of theory and epistemology [17]. These features made it possible to investigate related literatures while considering themes within the films and by comparing instances to data being examined for the other two studies.

A master catalog was created of the entire corpus of material. 20 films created over the course of a month were selected and transcribed for analysis. 18 were iteratively segmented and cataloged to isolate 68 instances of active error. 43 instances were thematically coded for evidence of error handling.

Taken together, many incidents are less than five minutes in length, though others span longer sequences of time. The longest spanned fifty minutes and two films. In most cases, incidents are resolved on camera within a single film. Each of the instances used in reporting below spanned a minute or two.

## III. Findings

Error handling is generally described as a three-stage process [18]. A person must know that an error has occurred, identify both what was "done wrong" and "what should have been done" and then understand how to "undo" the effects of the error [19, p. 476]. Handling unfolds in the course of "progressive" problem solving. An error is suspected or detected, and an evaluation is made to identify the source of the problem [20]. Environmental cues supply feedback to the problem solver by blocking forward progress [9], communicating about problems in system state [21] or by circumstantially guiding a problem solver to recovery [3].

Following this rubric, features of active error are illustrated in the following sections using statements and exchanges of dialogue between Marcus and Joe.

### A. Slips of Action

Actions sometimes do not go as planned, or were not intended. They are often simple, routine, and are commonly detected in the act based on perceptions that arise while doing something [19]. Often described in software engineering in terms of backtracking [13], they could also be described as slips of action [9]. Selecting the wrong item from a drop down menu or improperly referencing a variable are two examples:

***Marcus**: Oops, that's not what I want to do. (Ep. 12, 00:04:45)*

***Joe**: No can't do that cause it's. Oh we can move it outside the [try block ]… (Ep. 7, 00:06:51)*

In these cases, each developer gives a clear indication that something is wrong. What should have been done is evident, and recovery is simple. It is likely that Marcus caught his error in the act. Detection is also commonly made by assessing outcomes, and Joe's statement suggests that he may have responded to effects his actions had on the development environment.

### B. The Shape of Experience

Marcus and Joe are using an IDE, and follow principles of test-driven development [22]. Practice is error-directed: the pair write tests for intended behavior that initially fail, and are then proven within the implementation. In these circumstances, Marcus and Joe expect problems to be signaled by system responses [21]: red bars under method calls or arguments, error messages in the problems pane, or stack traces in the web browser.

Error handling is often required when conditions and situations are novel [23], when something comes up that has not been seen or done before. This is true even in the context of error-directed practice. Marcus and Joe may use and rely upon system responses to organize practice, but when an active error arises they are surprised and may be "stumped". They question outcomes [10], express uncertainty about how to proceed or communicate that they do not understand what is wrong.

Attention is often commanded because conditions are unexpected or new, but subsequent handling may draw upon knowledge gained through previous experience. The situation can turn out to be familiar. The following exchange demonstrates both perspectives:

***Marcus**: Now this is something to do, I had to solve this recently and I can't remember how I did it.*

***Joe**: It's an import, you need to import it, don't you? Or it needs to be umm, oh wait, its trying to execute that as a--*

***Marcus**: --It's the, the look. There's a, I did this before. It's to do with the way it does the test running stuff. (Ep. 2, 00:20:43)*

Joe makes three guesses about the source of the problem. Guessing is informal and pervasive within the catalog, used to direct handling. Guesses may be confident or timid, but are often wrong. Sometimes ideas are put forward that are partially informed, such Marcus' proposal about the source of the problem. However, guesses are also often made solely in response to behavior that is observed in the software. In Joe's case, they are an indicator of novelty, and suggest that he has encountered a problem that will require conscious problem solving [36].

In some handling processes, prior knowledge is well formed. It may even match the situation at hand so closely, it can be applied as a "recipe" or rule [24]:

***Marcus**: So we have a problem there...that I've noticed happens sometimes. If you actually stop it, now go back to Eclipse and stop it. And then start it again… (Ep. 1, 00:08:18)*

Recovery using the rule is straightforward. Marcus has seen the issue and is able to provide Joe with a mechanism for recovery. The solution is clear, but the circumstances surrounding the issue's earlier occurrence are unknown: Marcus does not indicate how difficult it was to solve, what was tried or how long it took.

### C. Forming Patterns of Practice

To understand how knowledge forms, it is necessary to compare data across instances. There is evidence in the catalog of the same error occurring in three different films that were made on different days. In each case, Marcus and Joe do not extend an exception class when it is created to satisfy conditions in a test. Here is what handling looks like the first time the error is signaled by a red bar:

***Joe**:...why is that complaining? Oh that's because we haven't got the constructors.*

***Marcus**: That's right.*

***Joe**: Oh, no, that's not, it says it's not a subtype of Exception. Oh [The class giving the error is opened]--*

***Marcus**: --'Cause it doesn't extend RuntimeException (Ep. 7, 00:02:57)*

Detection in this case is delayed, spurred during later practice when Joe tries to throw the exception. This kind of error could be interpreted as latent and analyzed deductively to determine the cognitive failure that led to its introduction in the code [10]. However, it is also possible to follow problem solving forward. Joe makes a guess about the source of the problem, signaling a shift from detection to identification and the pair undertake a brief cycle of local problem solving [3] to identify what is wrong.

The value of prospective analysis is made clearer by examining the subsequent occurrences. The second time a detection is made, the issue is familiar. Circumstances are slightly different; this time Marcus is adding a constructor to the exception class when a red bar appears. Joe is able to swiftly identify the source of the problem, and he takes responsibility for the error. He indicates that it might have been avoided:

***Joe**: Oh, that's 'cause it doesn't extend runtime. I was lazy and I didn't (inaudible).*

***Marcus**: But do you know what? Actually, ...I think now is the right time to actually put that in there. (Ep. 11, 00:16:53)*

Joe explains that the error was one of omission and that the class had not been created with strategic oversight [3]. However, Marcus counters that the omission is acceptable, because it upholds a preferred practice. This instance represents an example of the development of know-how or the formation of a rule-of-thumb. Rules in this sense are cultural [24], a point that is emphasized in these exchanges. The pair may be following principles of test-driven development and object oriented programming, but reserve the right to determine how classes are managed in relation to one another, even if this results in an error that reoccurs.

Joe's handling the third time enforces the practice and demonstrates the prior knowledge he has gained. There is no additional dialogue about how the error should be handled. It is still unexpected, but familiar, and handling has become routine. It is an error that can be caught more or less in the act and one that can be quickly recovered from using a rule.

***Joe**: Ahh [a red bar appears in the IDE]. So we didn't include the, when we created it we haven't made it extend exception. So now to make it... runtime exception. And we need a constructor with a message… (Ep. 18, 00:15:26)*

## IV. Related Work

In examining how developers find and fix things that go wrong, the findings presented here contribute to several existing strands of research.

Error handling has long been known to escape the confines of tools and processes associated with bugs. Root-cause analyses adopted a simplified definition of what errors are [2] in an effort to produce measurable improvements in software production [8]. Bug reports have been shown to be incomplete and inaccurate, with gaps of information that must be filled during practice through interaction [26]. Bug tracking tools are adapted to keep track of information about "almost bugs" [27], just as comments are used to track work that is ongoing [28]. Bugs are reassigned so developers that can be addressed by people who have active experience [29].

Errors become meaningful in terms of how they are described. In this study, analysis drew on Miyake's description of constructive interaction, an analytic that examines what people say when they solve problems together [30]. Mitigating the limitations associated with asking people to think-aloud [31], pairs undertaking tasks together naturally explain what they are thinking and give reasons for their ideas. The analytic can be employed in familiar environments, thereby producing dialogue and actions that reflect habits and patterns that are "typical" of a culture [13, p. 230].

Findings given in this report join other uses of the analytic that have examined how programmers use tools to restructure code [13], human computer interaction [32], collaboration [33] and team work [34]. Though they do not specifically cite constructive interaction as a methodological orientation, studies that examine pair programming likewise benefit from access to naturalistic exchanges of dialogue. Dialog-based verbalization is necessary during pair programming [35] and the activity has been studied for evidence of cognitive attributes like attention [36], and engagement [37].

The investigative process described here as local problem solving has been characterized in computing research as "bottom up". In the stories Eisenstadt gathered, developers did not systematically test hypotheses. Instead, they were found to have a rough idea of what they were looking for that they pursued in a process described as gathering data [38]. This

phenomenon has also been described as asking "Why?" [10], information seeking [39] or scent-following [40].

## V. Discussion

Error at the desk is not confined to activities that are normally associated with bugs. Errors arise when behavior is specified in tests, while classes are implemented, in periods when functionality is introduced and modified. They occur in relation to software that is being used and written and are primarily signaled by system responses. They are apparent: work is interrupted and developers clearly indicate that they are surprised.

As the examples show, error handling is often simple and compressed. However, sometimes recovery requires several rounds of local problem solving [3]. Guided by system responses [6], information gathering [38] is interspersed by manipulations of the environment. Mechanisms that might fix a problem are proposed at different points, sometimes more than once. Thus, though the successful removal of a system response is often noted, the fix itself often is not remarked upon at recovery.

Identification is enabled by how well developers assess and respond to circumstances in the immediate environment. The developers do not only read and respond to textual information or system responses. Assessments are also subtler: detection might be made if the layout of a page is different or if information designed into a system response by the developer is missing or incomplete.

Error handling can be prolonged. A single sequence of activity may represent the entire process, however some occurrences thread through the completion of other tasks. These issues invariably relate to "higher-order" concerns such as how to define conceptual boundaries for classes or how an object in a model should be expressed using features of a language. Incremental progress is made through verbal consensus that satisfies the developers and permits the issue to be set aside. In all cases, a subsequent instance occurs in which changes are made to the software.

The aim in handling is to get moving again, not to understand. This was demonstrated in the findings by juxtaposing how prior experience is used with novel experience. Joe did not need to understand why the mechanisms given to him by Marcus fixed the problem; he only needed to employ them. Likewise, partial understanding formed by Marcus during a prior experience was enough to direct a similar process that occurred later. The suggestion is given that gaps in understanding are acceptable and that fragments of knowledge are sufficient. Beyond acknowledging that something is "strange" or "weird", the developers do not always exhibit curiosity to learn more.

Recovery is not always permanent or complete. Errors are allowed to reoccur when they support a preferred practice. Circumstances may be similar, but handling will change as the issue becomes more familiar. Evidence is also given of issues in which handling is aborted. Severe incidents are unstable: investigations get out of hand, the developers indicate that they are lost or anxious, and that they find the process stressful.

Errors may be encountered together, but they are experienced alone. The findings demonstrate that an issue may be new to one developer and familiar to the other. Likewise, Marcus and Joe do not always notice that something has gone wrong at the same moment, or attribute the same significance to system responses or behaviors. Information is often freely given, but not received: the developer at the desk may not respond to suggestions given about actions to take or warnings about problems. At times, each developer appears to privilege behavior in the environment over what he is told, only making a detection once he can assess effects. Thus the same error may be caught in the act by one developer, but be detected based on outcomes by the other.

Working together serves error handling in several ways. Dialogue and commentary are important sources of feedback. Comments can focus a partner's attention, correct an assessment, or trigger an evaluation. The act of explaining a choice triggered detection in one case. Evidence was also given that pairs guide each other on occasion, dictating changes to be made in the code. Unlike the examples of recipes or rules given in the findings, the steps in these cases are not intended to produce a recovery for the error. Instead, they are given to stabilize the process, restoring immediate behavior so that problem solving can continue.

## VI. Limitations

This study performed detailed analysis on the actions of two developers. Findings are descriptive and may not extend to developers working in other circumstances. Data was drawn from secondary sources that were gleaned for data, and limitations on analysis were made by elements of the production. The camera depicts a limited view of activity, and it was necessary to account for gaps between tapings and technical difficulties in later films.

## VII. Conclusion

Error handling suffuses software development practice. Emphasis was given to error handling, the process undertaken to detect, identify and recover from an error. Findings demonstrate detections made in the act and based on outcomes, qualitative factors that influence identification, and how tracking error handling over time reveals details about how professional experience develops.

The meaning associated with an active error is personal. Its significance may diminish or develop, as a developer takes on new projects, in different environments and with different tools. Observations related to problem-solving and practice are consistent with findings in other software engineering research, but are revealed here to be representative more generally of human error as it has been conceived in other fields. Many additional questions may be asked of the data, and there are undoubtedly implications for tool development and methodology, two areas of work to be undertaken in the future.

## Acknowledgment

We thank the developers who informed this research.

## References


[1] .A. Avižienis, J. C. Laprie, and B. Randell, 'Dependability and its threats: a taxonomy', in *Building the Information Society*, vol. 156, R. Jacquart, Ed. Springer Boston, 2004, pp. 91–120.

[2] A. Endres, 'An analysis of errors and their causes in system programs', in *Proceedings of the International Conference on Reliable Software*, 1975, pp. 327–336.

[3] J. Reason, *Human Error*. New York: Cambridge University Press, 1990.

[4] J. Scott, *A matter of record: documentary sources in social research*, vol. 12. Polity Press Cambridge, 1990.

[5] B. Randell, 'On failures and faults', in *FME 2003: Formal Methods*, vol. 2805, K. Araki, S. Gnesi, and D. Mandrioli, Eds. Springer Berlin / Heidelberg, 2003, pp. 18–39.

[6] B. Randell, 'Dependability-a unifying concept', in *Proceedings of the Conference on Computer Security, Dependability, and Assurance: From Needs to Solutions*, 1998.

[7] S. Easterbrook, J. Singer, M. A. Storey, and D. Damian, 'Selecting empirical methods for software engineering research', *Guide to advanced empirical software engineering*, pp. 285–311, 2008.

[8] D. E. Perry, 'Where do most software flaws come from?', in *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson, Eds. O'Reilly Media, Inc., 2010, pp. 453–494.

[9] D. A. Norman, 'Categorization of action slips.', *Psychological review*, vol. 88, no. 1, pp. 1–15, 1981.

[10] A. J. Ko and B. A. Myers, 'A framework and methodology for studying the causes of software errors in programming systems', *Journal of Visual Languages & Computing*, vol. 16, no. 1, pp. 41–84, 2005.

[11] T. Lopez, M. Petre, and B. Nuseibeh, 'Getting at ephemeral flaws', in *Cooperative and Human Aspects of Software Engineering (CHASE), 2012 5th International Workshop*, 2012, pp. 90–92.

[12] J. Rasmussen, 'The role of error in organizing behaviour', *Ergonomics*, vol. 33, no. 10–11, pp. 1185–1199, 1990.

[13] R. W. Bowdidge and W. G. Griswold, 'How software engineering tools organize programmer behavior during the task of data encapsulation', *Empirical Software Engineering*, vol. 2, no. 3, pp. 221–267, 1997.

[14] H. Sharp, H. Robinson, and M. Woodman, 'Software engineering: community and culture', *Software, IEEE*, vol. 17, no. 1, pp. 40 –47, Feb. 2000.

[15] M. K. McGinn, 'Secondary data', in *The Sage encyclopedia of qualitative research methods*, L. M. Given, Ed. Sage Publications, 2008.

[16] M. Hammersley and P. Atkinson, *Ethnography: Principles in practice*. Routledge, 2007.

[17] V. Braun and V. Clarke, 'Using thematic analysis in psychology', *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[18] F. C. Brodbeck, D. Zapf, J. Prümper, and M. Frese, 'Error handling in office work with computers: A field study', *Journal of occupational and organizational psychology*, vol. 66, no. 4, pp. 303–317, 1993.

[19] A. J. Sellen, 'Detection of everyday errors', *Applied Psychology*, vol. 43, no. 4, pp. 475–498, 1994.

[20] C. M. Allwood, 'Error detection processes in statistical problem solving', *Cognitive science*, vol. 8, no. 4, pp. 413–437, 1984.

[21] C. Lewis and D. A. Norman, 'Designing for Error', in *User Centered System Design*, Erlbaum Associates, Inc., 1986.

[22] S. Ambler, 'Introduction to test driven development', 2012. [Online]. Available: http://www.agiledata.org/essays/tdd.html.

[23] D. A. Norman and T. Shallice, 'Attention to action', in *Consciousness and Self-Regulation*, R. J. Davidson, G. E. Schwartz, and D. Shapiro, Eds. Springer US, 1986, pp. 1–18.

[24] J. Rasmussen, 'Human error data. Facts or fiction?', Riso National Laboratory, Roskilde, Denmark, 1985.

[25] Y. Dittrich, D. W. Randall, and J. Singer, 'Software engineering as cooperative work', *Computer Supported Cooperative Work*, vol. 18, no. 5–6, pp. 393–399, 2009.

[26] J. Aranda and G. Venolia, 'The secret life of bugs: going past the errors and omissions in software repositories', in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 298–308.

[27] D. Bertram, A. Voida, S. Greenberg, and R. Walker, 'Communication, collaboration, and bugs: The social nature of issue tracking in software engineering', in *Proc. ACM Conf. Comput. Support. Coop. Work*, 2010.

[28] M. A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer, 'TODO or to bug', in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, 2008, pp. 251–260.

[29] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, 'Not my bug! and other reasons for software bug report reassignments', in *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, 2011, pp. 395–404.

[30] N. Miyake, 'Constructive interaction and the iterative process of understanding', *Cognitive Science*, vol. 10, no. 2, pp. 151–177, 1986.

[31] J. Hughes and S. Parkes, 'Trends in the use of verbal protocol analysis in software engineering research', *Behaviour & Information Technology*, vol. 22, no. 2, pp. 127–140, 2003.

[32] D. Wildman, 'Getting the most from paired-user testing', *interactions*, vol. 2, no. 3, pp. 21–27, Jul. 1995.

[33] N. V. Flor, 'Side-by-side collaboration: A case study', *International Journal of Human-Computer Studies*, vol. 49, no. 3, pp. 201–222, 1998.

[34] N. V. Flor and E. L. Hutchins, 'A case study of team programming during perfective software maintenance', in *Empirical studies of programmers: Fourth workshop*, 1991, p. 36.

[35] S. Xu and V. Rajlich, 'Dialog-based protocol: an empirical research method for cognitive activities in software engineering', in *Empirical Software Engineering, 2005. 2005 International Symposium on*, 2005, p. 10–pp.

[36] A. Sillitti, G. Succi, and J. Vlasenko, 'Understanding the impact of pair programming on developers attention: a case study on a large industrial experimentation', in *Proceedings of the 2012 International Conference on Software Engineering*, 2012, pp. 1094–1101.

[37] L. Plonka, H. Sharp, and J. Van der Linden, 'Disengagement in pair programming: does it matter?', in *Software Engineering (ICSE), 2012 34th International Conference on*, 2012, pp. 496–506.

[38] M. Eisenstadt, 'My hairiest bug war stories', *Communications of the ACM*, vol. 40, no. 4, pp. 30–37, 1997.

[39] A. J. Ko, R. DeLine, and G. Venolia, 'Information needs in collocated software development teams', in *Proceedings of the 29th international conference on Software Engineering*, 2007, pp. 344–353.

[40] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, 'How programmers debug, revisited: An information foraging theory perspective', *Software Engineering, IEEE Transactions on*, vol. 39, no. 2, pp. 197–215, 2013.