10-2023

Human Error Assessment in Software Engineering

Benjamin S. Meyers
bsm9339@rit.edu

Human Error Assessment in Software Engineering

by

Benjamin S. Meyers

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York
October 2023

Human Error Assessment in Software Engineering

by

Benjamin S. Meyers

Committee Approval:
We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

Dr. Andrew Meneely Dissertation Advisor	Date
Dr. Daniel Krutz Dissertation Committee Member	Date
Dr. Mehdi Mirakhorli Dissertation Committee Member	Date
Dr. Emily Prud'hommeaux Dissertation Committee Member	Date
Dr. Sharon Mason Dissertation Defense Chairperson	Date

Certified by:

Dr. Pengcheng Shi Ph.D. Program Director, Computing and Information Sciences	Date

Human Error Assessment in Software Engineering

by

Benjamin S. Meyers

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences Ph.D. Program in Computing and Information Sciences
in partial fulfillment of the requirements for the
Doctor of Philosophy Degree
at the Rochester Institute of Technology

Abstract

Software engineers work under strict constraints, balancing a complex, multi-phase development process on top of user support and professional development. Despite their best efforts, software engineers experience **human errors**, which manifest as software defects. While some defects are simple bugs, others can be costly security vulnerabilities. Practices such as defect tracking and vulnerability disclosure help software engineers reflect on the outcomes of their human errors (*i.e.* software failures), and even the faults that led to those failures, but not the underlying human behaviors. While human error theory from psychology research has been studied and applied to medical, industrial, and aviation accidents, researchers are only beginning to systematically reflect on software engineers' human errors. Some software engineering research has used human error theories from psychology to help developers identify and organize their human errors (mistakes) during requirements engineering activities, but developers need an improved and systematic way to reflect on their human errors during other phases of software development. The goal of this dissertation is ***to help software engineers confront and reflect on their human errors by creating a process to document, organize, and analyze human errors***. To that end, our research comprises three phases: (1) systematization (*i.e.* identification and taxonomization) of software engineers' human errors from literature and development artifacts into a Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.), (2) evaluation and refinement of T.H.E.S.E. based on software engineers' perceptions and natural language insights, and (3) creation of a human error informed micro post-mortem process and the Human Error Reflection Engine (H.E.R.E.), a proof-of-concept GitHub workflow facilitating human error reflection. In demonstrating the utility of T.H.E.S.E. and our micro post-mortem process, the software development community will be closer to inculcating the wisdom of historical developer human errors, enabling them to engineer higher quality and more secure software.

Acknowledgments

This dissertation is the culmination of 23 years of schooling, and the first milestone in my life-long journey of discovery. The journey so far would not have been possible without the help and support of the following individuals.

First and foremost, I need to thank my advisor, **Dr. Andrew Meneely**, for believing in me, for pushing me to meet my potential, for making sure I always had funding, for advocating on my behalf, for guiding me along the way, and for his *abundant* patience. Many thanks to my dissertation committee—**Dr. Daniel Krutz**, **Dr. Mehdi Mirakhorli**, and **Dr. Emily Prud'hommeaux**—for their guidance, support, and direction. Thanks to **Dr. Pengcheng Shi** for early encouragement. Special thanks to **Mr. Kirk Anne**, **Dr. Omar Aponte**, and **Dr. Tre DiPassio** for thoughtfully reviewing all 200 pages of my dissertation. I must also thank everyone who worked behind the scenes to help me succeed—**Ms. Min-Hong Fu**, **Mr. Charles Gruener**, **Ms. Tracy Miller**, **Ms. Sarah Mittiga**, **Mr. Kurt Mosiejczuk**, **Ms. Chelsea O'Brien**, **Ms. Dawn Smith**, **Ms. Lorrie Jo Turner**, **Ms. Siyuan Wang**, and **Ms. Amanda Zeluff**.

As a life-long student, I have many educators to thank. Thanks to **Ms. Andrea Chochul** for helping a struggling 2nd grader become an excellent reader. Thanks to **Ms. Jennifer Case** for helping an 8th grader foster a passion for reading. Thanks to **Mr. Timothy Strohm** for helping an 11th grader steer his future towards software engineering and education. Thanks to **Mr. Jason Rees** and the many mentors of **F.I.R.S.T. Robotics Team 340**—notably **Mr. John Lawniczak**, **Mr. Rob Heslin**, **Mr. Rex Hays**, **Mr. Gary Stafford**, and **Ms. Ellen Swift**—for their countless hours and commitment to my extracurricular education. Thanks to **Mr. John Andersen** for helping a frustrated undergraduate finally pass physics. Thanks to **Mr. Kenn Martinez** for his encouragement and wisdom. Thanks to **Dr. Ernest Fokoué** for his contagious sense of wonder. Thanks to **Dr. Naveen Sharma** for the opportunity to teach SWEN-331: Engineering Secure Software. Thanks to **Ms. Marie Barron**, **Mr. Karl Biedlingmaier**, **Dr. Zhong Chen**, **Dr. Travis Desell**, **Ms. Lynn Dimbleby**, **Mr. James Hauck**, **Dr. Thomas Kinsman**, **Mr. Larry Kiser**, **Dr. Minseok Kwon**, **Ms. Laura Mahar**, **Dr. Yin Pan**, **Mr. Joseph Pencille**, **Mr. Kal Rabb**, **Mr. Tom Reichlmayr**, **Mr. Wilson Silva**, **Ms. Nancy Thornton**, **Dr. James Vallino**, **Ms. Cheryl Valvano**, **Dr. Linwei Wang**, and **Dr. Matthew Wright**.

Thanks to my **SWEN-331 students**—Alan X., Alex L., Alex N., Alex P., Aly S., Andrew H., Arian J., Asha H., Bailey G., Celeste G., CJ S., Daniel B., Dennis N., Fahd M., Haeri K., Hunter K., Jackson K., Jake O., Jared P., Jason H., Jennika H., Jimmy D., John A., Jonah F., Joseph D., Kyle M., Logan P., Lukas S., Mallory B., Matt R., Max K., Meghan J., Michael B., Michael D., Michael K., Milo B., Nicholas A., Nicholas B., Nick S., Noah B., Owen R., Patrick D., Pedro B., Peter A., Prionti N., Ryan G., Samuel V., Sean B., Sergey G., Shane B., Spencer A., Spencer F., Suky K., Timothy M., Tyler B., Tyler C. —I hope you learned as much from me as I learned from you.

I have had the pleasure of working with and learning from many researchers. Special thanks to **Dr. Emily Prud'hommeaux** and **Dr. Cecilia O. Alm** for sparking my passion for linguistics, and for taking a chance on an undergraduate and giving me my first taste of research. Special thanks to **Dr. Nuthan Munaiah** for taking me under his wing during my formative Ph.D. years. Special thanks to **Brandon N. Keller** for his contributions to the studies discussed in Chapter 4. Thanks to **Sultan Fahad Almassari**, **Ryan Cervantes**, **Dr. Cristian Danescu-Niculescu-Mizil**, **Dr. Klaus Greve**, **Dr. Edward Hensel**, **Brienna Herold**, **Chris Horn**, **Dr. Matt Huenerfauth**, **Mr. Samuel Malachowsky**, **Dr. Sharon Mason**, **Dr. Mohamed Wiem Mkaouer**, **Dr. Elizabeth Moore**, **Dr. Stephen Moskal**, **Dr. Pradeep Murukannaiah**, **Dr. Christian Newman**, **Kyle Parnell**, **Dr. Justin Pelletier**, **Dr. Esa Rantanen**, **Dr. Jörg Szarzynski**, **Dr. Brian Tomaszewski**, **Dr. Laurie Williams**, **Dr. Josephine Wolff**, **Dr. Jay Yang**, and **Dr. Yang Yu**.

I cannot thank Research Computing [312]—**Kirk Anne**, **Emilio Del Plato**, **Andrew Elble**, **Douglas Heckman**, **Jennifer Herting**, **Paul Mezzanini**, **Sidney Pendelberry**—enough for their assistance, mentorship, employment, friendship, and constant entertainment.

I wish to thank **my family** for their support, even though they rarely understood what I was talking about. Many thanks to my friends for providing some sense of normalcy throughout my dissertation—**CJ**, **Brittany**, **Tre**, **Miranda**, **Nikki**, **Michael**, **Katya**, **Paige**, **Dan**, **Darbi**, **Andy**, **Jordan**, **Jens**, **Adam**, **Grant**, the **Doctoral Student Association**—notably **Omar**, **Cyril**, and **Sam**— **Oliver**, **Brett**, **Jay**, **Joanna**, **Ed**, **Richard**, the **Atene family**, the **Del Plato family**, the **Evans family**, and the **Mezzanini family**. Special thanks to my girlfriend, **Chloe**, for her encouragement and patience during the last few months of my dissertation.

Special thanks to my therapist for helping me manage the stress of being a Ph.D. student. Additional thanks to my many unofficial *therapy pets*—Astro, Smokey, Tucker, Stella, Lily, Aglio, Orville, Cheechie, Ziti, Coco, Muppet, Mew-Mew, Blackbeard, Sausage, Kitten, Mama, Biggie, Tupac, KitKat, Oly, Maeve, Wren, Tulip, Bagel, Goose, Cereal, Minnow, Bunny, the rats, the axolotls, and even Gary (you jerk).

Thanks to the following scientists, educators, artists, and authors for inspiring me without ever knowing me—Douglas Adams, Hank & John Green, Jack Kirby, Stan Lee, Michael Pollan, James Reason, Fred Rogers, Carl Sagan, Joe Shuster, Jerry Siegel, Paul Stamets, Nikola Tesla, J.R.R. Tolkien, Neil deGrasse Tyson, and David Wallace-Wells.

Our research has been supported by the U.S. National Security Agency (NSA) Science of Security Lablet (SoSL) at North Carolina State University (H98230-17-D-0080/2018-0438-02), the National Science Foundation (grant 1922169), and the Small Business Innovation Research (SBIR) Program of the U.S. Defense Advanced Research Projects Agency (DARPA) (grant 140D63-19-C-0018).

"To err is human but a human error is nothing to what a computer can do if it tries."

Agatha Christie, Author

"The difference between us and a computer is that, the computer is blindingly stupid, but it is capable of being stupid many, many million times a second."

Douglas Adams, Author

"At the source of every error which is blamed on the computer, you will find at least two human errors, one of which is the error of blaming it on the computer."

Tom Gilb, Software Engineer

"Children aren't afraid to try things and make mistakes because you learn from those. We grown-ups must learn from experience, of course, but we can't be afraid to do something and risk making a mistake. We've got to remember that we can keep learning throughout our entire lives. And play our entire lives."

Kjeld Kirk Kristiansen, Former Owner of Lego

"A young apprentice applied to a master carpenter for a job. The older man asked him, 'Do you know your trade?' 'Yes, sir!' the young man replied proudly. 'Have you ever made a mistake?' the older man inquired. 'No, sir!' the young man answered, feeling certain he would get the job. 'Then there's no way I'm going to hire you,' said the master carpenter, 'because when you make one, you won't know how to fix it.'"

Mister Fred Rogers, Neighbor to All

"The only real mistake is the one from which we learn nothing."

Henry Ford, Innovator

"Mistakes are the portals of discovery."

James Joyce, Poet

"To the scientist, the universality of physical laws makes the cosmos a marvelously simple place. By comparison, human nature—the psychologist's domain—is infinitely more daunting."

Neil deGrasse Tyson, Science Communicator

"Think of people as people, not problems that need to be solved."

Hank Green, Science Communicator

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software development is a complex process of gathering requirements from stakeholders, designing a software system, and implementing, testing, deploying, and maintaining that software. Each stage of the software development lifecycle encompasses many activities intended to improve software quality—risk assessment, threat modeling, code review, a wide range of testing, responding to bug reports, patching vulnerabilities, and deploying updates—in addition to communicating with peer developers, managers, and stakeholders. No complex process is inherently safe [76], so software developers, despite their best efforts, inevitably make mistakes.

Mistakes, or **human errors**, have been extensively documented and studied in psychology research for well over 50 years. Sigmund Freud drew attention to human error in 1901 as he studied slips of the tongue, forgetfulness, and omissions [105]. In 1937, Kollarits conducted one of the first studies of human error, examining about 1,200 human errors experienced by himself, his wife, and colleagues. Kollarits arrived at four categories of human error: substitution, omission, repetition, and insertion [169]. Later in the 1980s, Jens Rasmussen studied human error primarily in the context of industrial accidents [294, 296], and classified human errors as skill-, rule-, or knowledge-based [293, 295]. Building on Rasmussen's work, James Reason categorized human errors under his Generic Error-Modelling System (GEMS) [298] as slips, lapses, and mistakes (failures of attention, memory, and planning, respectively).

Software engineers and security practitioners have also been concerned with human errors (albeit indirectly), defining, studying, and cataloging the consequences of software developers' human errors—defects (faults and failures), weaknesses, bugs, and vulnerabilities. With open source software, software engineers have documented their mistakes and put them on display, adopting vulnerability disclosure as an industry standard so that current and future developers can learn from their mistakes. The Common Vulnerabilities and Exposures (CVE) database [237], for example, provides a timestamp and a brief description of thousands of documented vulnerabilities, along with links to related security advisories, changelogs, bug reports, patches, and severity measures. Further, the Common Weakness Enumeration (CWE) [235] taxonomizes vulnerabilities into technical faults and their mitigations. While the CVE and CWE are not perfect, they are systematic endeavors which enable learning from critical software security mistakes.

Vulnerability assessment is a noble and valuable endeavor, but it's only one piece of the puzzle: human errors lead to software faults [349]. However, adoption of human error research from psychology into software development has been relatively slow and short-lived (except for work by Anu *et al.* [12, 15, 18, 22, 23]). While some software developers and researchers have paid attention to human error, Wood & Banks expressed concern in 1993 that "many contemporary information security practitioners appear to have forgotten about [362]" human error, and 12 years later Im & Baskerville expressed "the serious need for research and practical knowledge about the management of human error in secure information systems [147]." Human error is particularly impactful in software engineering (a process-oriented domain) because

> *"people are key components of processes. They are involved in process design, operation, maintenance, etc.* ***No step in the process life cycle is without some human involvement. Based on human nature, human error is a given and will arise in all parts of the process life cycle.*** *Also,* ***processes are generally not well-protected from human errors*** *since many safeguards are focused on equipment failure. Consequently, it is likely that human error will be an important contributor to risk for most processes [35]."* (emphasis added)

Examining faults and failures on their own does not allow software engineers to assess their human errors; software engineers need to dig deeper and consider the underlying human errors behind their software defects. Figure 1.1 shows conceptually how developers' human errors have a lasting impact on software development, leading to faults in code,

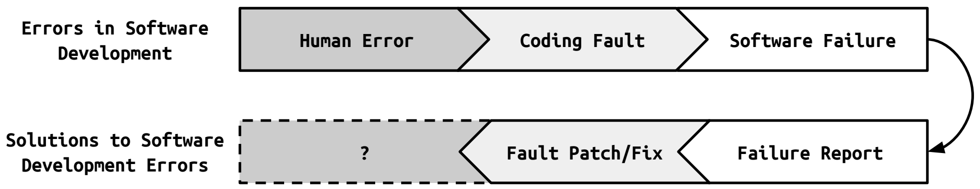

Errors in Software Development
Human Error
Coding Fault
Software Failure
Solutions to Software Development Errors
?
Fault Patch/Fix
Failure Report

Figure 1.1: Lifecycle of Human Error in Typical Software Engineering
Each stage of error in software development has its own solution. Software failures are addressed with failure reports and their underlying faults are patched/fixed. However, we are unaware of any standard software engineering process/solution for addressing the human errors leading to faults.

which manifest as software failures [350]. Failures are reported, and faults patched, but to our knowledge, typical software engineering does not include evidence-based human error assessment. The goal of this work is ***to help software engineers confront and reflect on their human errors by creating a process to document, organize, and analyze human errors***. To that end, this dissertation comprises three phases:

Phase 1: Systematization (*i.e.* identification and taxonomization) of software engineers' human errors from literature and development artifacts into a Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.)

Phase 2: Evaluation of T.H.E.S.E. based on software engineers' perceptions and natural language insights

Phase 3: Creation of a human error informed micro post-mortem process and the Human Error Reflection Engine (H.E.R.E.), a proof-of-concept GitHub workflow facilitating human error reflection

At this point, we must reassure the reader that the goal of our work is **not** to provide a framework for placing blame on individual software engineers. Blaming humans for the human errors they experience is the old view of human error; the new view is that human errors are a symptom of environmental factors and constraints [76]. Software engineers

> *"confront different evolving situations, they navigate and negotiate the messy details of their practice to bridge gaps and to join together the bits and pieces of their system, creating success as a balance between the multiple conflicting goals and pressures imposed by their organizations [363]."*

Human errors are not the result of malice[1] on the part of the software engineer, but an unavoidable part of the process. Our goal is not to place blame or remove human error altogether, but to provide a process for software engineers to document and reflect on their experienced human errors so they can implement mitigations and process improvements, and help future developers avoid similar human errors.

1.1 Overview

In this section, we provide an overview of the remaining chapters in this dissertation. Chapter 2 provides a brief background on concepts related to human error, natural language processing, software engineering, computer security, and taxonomies. In Chapter 3, we outline related work on human error and taxonomy development in software engineering.

In Chapter 4, we describe the first phase of this dissertation, which consists of two research studies. First, we conducted a systematic literature review (SLR) of 68 research studies about human errors in software engineering. Using established terminology and multiple taxonomies from existing psychology research, we systematically examined a subset of 81,007 search results from three research paper databases, and aggregated software engineers' human errors (192 in total) into Version 1 of our Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.). Next, we collected 88.6 million developer comments from GitHub. Using apology lemmas from linguistics research, we created an automatic apology classification process. We manually categorized a random subset of 332 apology comments according to the experienced human error. We compared human error categories derived from software engineers' apologies with Version 1 of T.H.E.S.E. and updated T.H.E.S.E. (Version 2) with categories of human error in software engineering that were not previously documented in literature. These two studies address the following research questions:

[1]See Section 2.1.2 for a discussion of *violations*.

RQ 1: Human Error Discovery
What human errors experienced by software engineers are documented in previous research?

RQ 2: Human Error Appraisal
What are the strengths and weaknesses of previous research about human error in software engineering? We define strengths and weaknesses in terms of software engineering scope, category ambiguity, and category completeness.

RQ 3: Human Error Usefulness
How comprehensive is existing research about human errors in software engineering? We define usefulness in terms of coverage of human error theories from cognitive psychology and coverage of human errors in software engineering activities.

RQ 4: Identifying Developers' Apologies
Can apology lemmas reliably identify developers' apologies in development artifacts?

RQ 5: Anatomy of Developers' Apologies
How often do developers apologize and which apology lemmas are most common?

RQ 6: Developers' Self-Admitted Human Errors
Which human errors from literature do developers admit to? Which human errors from developer apologies do not exist in literature?

In the second phase of this dissertation (Chapter 5), we evaluated T.H.E.S.E. with two approaches. First, we conducted a user study in which software engineering students at the Rochester Institute of Technology were provided training on human error and T.H.E.S.E., then tasked with documenting and categorizing their human errors. Throughout the user study, participants met weekly with the author of this dissertation to discuss their human errors and categorization further. We used insights from the user study to improve T.H.E.S.E. category titles and definitions (Version 3 of T.H.E.S.E.), as well as design a formal micro post-mortem process for human error assessment in software engineering. Next, we refined T.H.E.S.E. using pretrained natural language models. Based on classification results, we further improved T.H.E.S.E. category titles and definitions (Version 4 of T.H.E.S.E.). Phase 2 of this dissertation addresses the following research questions:

RQ 7: Ease of Use
How clear, unambiguous, and simple is T.H.E.S.E. for software engineers to use?

RQ 8: Comprehensiveness
How well does T.H.E.S.E. cover human errors in software engineering?

RQ 9: Assessment Value
How well does T.H.E.S.E. facilitate human error reflection?

RQ 10: Category Ambiguity
How well can ambiguity of T.H.E.S.E. category descriptions be reduced based on semantic similarity?

RQ 11: Assisted Categorization
How useful is Sentence-BERT for refining human error definitions?

For the third and final phase of this dissertation, Chapter 6 describes (1) a formal human error informed micro post-mortem process for software engineering, and (2) a proof-of-concept Human Error Reflection Engine (H.E.R.E.) to facilitate micro post-mortems on GitHub. In Chapter 7, we summarize our work and provide recommendations for future work.

1.2 Contributions

This dissertation makes the following contributions to the field of software engineering.

1.2.1 Publications

Throughout my academic career (both undergraduate and graduate), I have contributed to 10 research publications, which are listed below. Those that do not directly relate to this dissertation—marked with an asterisk (*)—are included as they eventually led me to my dissertation topic. Those that are currently undergoing peer-review are denoted by a dagger (†):

1. **Natural Language Insights from Code Reviews that Missed a Vulnerability.* (2017). Nuthan Munaiah, **Benjamin S. Meyers**, Cecilia O. Alm, Andrew Meneely, Pradeep K. Murukannaiah, Emily Prud'hommeaux, Josephine Wolff, and Yang Yu. Proceedings of the 9th International Symposium for Engineering Secure Software and Systems *(ESSoS)*. Bonn, Germany. [245]

   This study examined the characteristics of software vulnerabilities that were missed. Since there is human error behind every bug and vulnerability [349], this work indirectly fueled my interest in software engineers' human errors.

   Abstract: Engineering secure software is challenging. Software development organizations leverage a host of processes and tools to enable developers to prevent vulnerabilities in software. Code reviewing is one such approach which has been instrumental in improving the overall quality of a software system. In a typical code review, developers critique a proposed change to uncover potential vulnerabilities. Despite best efforts by developers, some vulnerabilities inevitably slip through the reviews. In this study, we characterized linguistic features—inquisitiveness, sentiment and syntactic complexity—of conversations between developers in a code review, to identify factors that could explain developers missing a vulnerability. We used natural language processing to collect these linguistic features from 3,994,976 messages in 788,437 code reviews from the Chromium project. We collected 1,462 Chromium vulnerabilities to empirically analyze the linguistic features. We found that code reviews with lower inquisitiveness, higher sentiment, and lower complexity were more likely to miss a vulnerability. We used a Naïve Bayes classifier to assess if the words (or lemmas) in the code reviews could differentiate reviews that are likely to miss vulnerabilities. The classifier used a subset of all lemmas (over 2 million) as features and their corresponding TF-IDF scores as values. The average precision, recall, and F-measure of the classifier were 14%, 73%, and 23%, respectively. We believe that our linguistic characterization will help developers identify problematic code reviews before they result in a vulnerability being missed.

2. **An Analysis and Visualization Tool for Case Study Learning of Linguistic Concepts.* (2017). Cecilia O. Alm, **Benjamin S. Meyers**, and Emily Prud'hommeaux. Proceedings of the Conference on Empirical Methods in Natural Language Processing *(EMNLP)*. Copenhagen, Denmark. [7]

   This study was my first introduction to pedagogy and my first experience designing learning tools. T.H.E.S.E. is ultimately a tool for learning, and I credit this work with steering my research in that direction.

   Abstract: We present an educational tool that integrates computational linguistics resources for use in non-technical undergraduate language science courses. By using the tool in conjunction with evidence-driven pedagogical case studies, we strive to provide opportunities for students to gain an understanding of linguistic concepts and analysis through the lens of realistic problems in feasible ways. Case studies tend to be used in legal, business, and health education contexts, but less in the teaching and learning of linguistics. The approach introduced also has potential to encourage students across training backgrounds to continue on to computational language analysis coursework.

3. **A Dataset for Identifying Actionable Feedback in Collaborative Software Development.* (2018). **Benjamin S. Meyers**, Nuthan Munaiah, Emily Prud'hommeaux, Andrew Meneely, Cecilia O. Alm, Josephine Wolff, and Pradeep Murukannaiah. Proceedings of the 2018 Meeting for the Association for Computational Linguistics *(ACL)*. Melbourne, Australia. [218]

   This study was a valuable experience orchestrating the mining of a large natural language dataset. The skills established during this study were invaluable in mining software engineers' apologies in Section 4.2.

   Abstract: Software developers and testers have long struggled with how to elicit proactive responses from their coworkers when reviewing code for security vulnerabilities and errors. For a code review to be successful, it must not only identify potential problems but also elicit an active response from the colleague responsible for modifying the code. To understand the factors that contribute to this outcome, we analyze a novel dataset of more than one million code reviews for the Google Chromium project, from which we extract linguistic features of feedback that elicited responsive actions from coworkers. Using a manually-labeled subset of reviewer comments, we trained a highly accurate classifier to identify "acted-upon" comments (AUC = 0.85). Our results demonstrate the utility of our dataset, the feasibility of using NLP for this new task, and the potential of NLP to improve our understanding of how communications between colleagues can be authored to elicit positive, proactive responses.

4. **Pragmatic Characteristics of Security Conversations: An Exploratory Linguistic Analysis.* (2019). **Benjamin S. Meyers**, Nuthan Munaiah, Andrew Meneely, and Emily Prud'hommeaux. Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering *(CHASE)*. Montréal, QC, Canada. [219]

    This study further honed my data mining and statistics experience, while also inspiring me to think about the variety of things we can learn from software engineering artifacts.

    Abstract: Experts suggest that engineering secure software requires a defensive mindset to be ingrained in developer culture, which could be reflected in conversation. But what does a conversation about software security in a real project look like? Linguists analyze a wide array of characteristics: lexical, syntactic, semantic, and pragmatic. *Pragmatics* focus on identifying the style and tone of the author's language. If security requires a different mindset, then perhaps this would be reflected in the conversations' pragmatics. Our goal is to characterize the pragmatic features of conversations about security so that developers can be more informed about communication strategies regarding security concerns. We collected and annotated a corpus of conversations from 415,041 bug reports in the Chromium project. We examined five linguistic metrics related to pragmatics: formality, informativeness, implicature, politeness, and uncertainty. Our initial exploration into these data show that pragmatics plays a role, however small, in security conversations. These results indicate that the area of linguistic analysis shows promise in automatically identifying effective security communication strategies.

5. **Developing a Geographic Information Capacity (GIC) Profile for Disaster Risk Management Under United Nations Framework Commitments.* (2020). Brian M. Tomaszewski, Elizabeth A. Moore, Kyle Parnell, Alexandra M. Leader, William R. Armington, Omar Aponte, Leslie Brooks, Brienna K. Herold, **Benjamin S. Meyers**, Tayler Ruggero, Zachary Sutherby, Madeline Wolters, Sandy Wua, Jörg Szarzynski, Klaus Greve, and Robert Parody. International Journal of Disaster Risk Reduction *(IJDRR)*. [341]

    While unrelated to software engineering, this study examined risk factors and remediations. Software engineers frequently assess risk, and human error management (T.H.E.S.E. could be considered a human error management tool) is itself a form of risk management.

    Abstract: The capacity to utilize geographic information is a critical element of disaster risk management. Although access to and use of geographic information system (GIS) technology continues to grow, there remain significant gaps in approaches used by disaster risk management stakeholders to understand geographic information needs, sources, and information flow—ultimately limiting the efficacy of management efforts. To address this problem, we introduce the concept of geographic information capacity (GIC) to measure and analyze the ability of stakeholders to understand, access, and work with geographic information for disaster risk management. We propose a framework for assessing GIC, the GIC Profile, which we situate within a review of disaster risk management-relevant frameworks. We evaluate the GIC Profile using two case study countries at the first (sub-national) geo-administrative boundary level. Chi-square analyses suggest GIC across equivalent regional units within each country is relatively uniform, and that this uniformity is comparable between nations despite significant difference in overall capacity. Contributions of the GIC Profile to disaster risk management research are twofold. First, this is a first attempt to develop a profile based on key indicators for quantifying GIC highlights critical areas for capacity improvement, allowing decision makers to identify and prioritize pathways to strengthen disaster risk management programs. Through this initial effort, a decision tool has been developed which may enhance decisions on how to utilize GIS in support of disaster risk management. This tool is iterative and can be updated as new events occur to maximize GIS benefits, ultimately reducing disaster risks and their potential consequences.

6. **An Automated Post-Mortem Analysis of Vulnerability Relationships using Natural Language Word Embeddings.* (2021). **Benjamin S. Meyers** and Andrew Meneely. Proceedings of the 3rd International Symposium on Machine Learning and Big Data Analytics for Cybersecurity and Privacy at *(ANT/EDI40)*. Warsaw, Poland (virtual due to COVID-19). [213]

This study involved identifying relationships between software vulnerabilities using word embeddings (which motivated our use of Sentence-BERT in Section 5.3) and continued my interest in learning from software engineers' *mistakes*.

Abstract: The daily activities of cybersecurity experts and software engineers—code reviews, issue tracking, vulnerability reporting—are constantly contributing to a massive wealth of security-specific natural language. In the case of vulnerabilities, understanding their causes, consequences, and mitigations is essential to learning from past mistakes and writing better, more secure code in the future. Many existing vulnerability assessment methodologies, like CVSS, rely on categorization and numerical metrics to glean insights into vulnerabilities, but these tools are unable to capture the subtle complexities and relationships between vulnerabilities because they do not examine the nuanced natural language artifacts left behind by developers. In this work, we want to discover unexpected relationships between vulnerabilities with the goal of improving upon current practices for post-mortem analysis of vulnerabilities. To that end, we trained word embedding models on two corpora of vulnerability descriptions from Common Vulnerabilities and Exposures (CVE) and the Vulnerability History Project (VHP), performed hierarchical agglomerative clustering on word embedding vectors representing the overall semantic meaning of vulnerability descriptions, and derived insights from vulnerability clusters based on their most common bigrams. We found that (1) vulnerabilities with similar consequences and based on similar weaknesses are often clustered together, (2) clustering word embeddings identified vulnerabilities that need more detailed descriptions, and (3) clusters rarely contained vulnerabilities from a single software project. Our methodology is automated and can be easily applied to other natural language corpora. We release all of the corpora, models, and code used in our work.

7. **Examining Penetration Tester Behavior in the Collegiate Penetration Testing Competition.* (2022). **Benjamin S. Meyers**, Sultan Fahad Almassari, Brandon N. Keller, and Andrew Meneely. ACM Transactions on Software Engineering and Methodology *(TOSEM).* [220]

   This study provided an unique experience to learn about software engineers' *mistakes* from the perspective of an attacker. This study provided my first experience systematizing software engineers' *mistakes*, a skill set that was invaluable in Chapter 4.

   Abstract: Penetration testing is a key practice toward engineering secure software. Malicious actors have many tactics at their disposal, and software engineers need to know what tactics attackers will prioritize in the first few hours of an attack. Projects like MITRE ATT&CK™ provide knowledge, but how do people actually deploy this knowledge in real situations? A penetration testing competition provides a realistic, controlled environment with which to measure and compare the efficacy of attackers. In this work, we examine the details of vulnerability discovery and attacker behavior with the goal of improving existing vulnerability assessment processes using data from the 2019 Collegiate Penetration Testing Competition (CPTC). We constructed 98 timelines of vulnerability discovery and exploits for 37 unique vulnerabilities discovered by ten teams of penetration testers. We grouped related vulnerabilities together by mapping to Common Weakness Enumerations and MITRE ATT&CK™. We found that (1) vulnerabilities related to improper resource control (*e.g.* session fixation) are discovered faster and more often, as well as exploited faster, than vulnerabilities related to improper access control (*e.g.* weak password requirements), (2) there is a clear process followed by penetration testers of discovery/collection to lateral movement/pre-attack. Our methodology facilitates quicker analysis of vulnerabilities in future CPTC events.

8. **What Happens When We Fuzz? Investigating OSS-Fuzz Bug History.* (2023). Brandon N. Keller, **Benjamin S. Meyers**, and Andrew Meneely. International Conference on Mining Software Repositories *(MSR).* [160]

   While mitigations for human errors in software engineering are outside the scope of my dissertation, tools are often suggested countermeasures for human error [285]. This study explored fuzzers, an often ignored type of tool aimed at assisting software engineers.

   Abstract: Software engineers must be vigilant in preventing and correcting vulnerabilities and other critical bugs. In servicing this need, numerous tools and techniques have been developed to assist developers. Fuzzers, by autonomously generating inputs to test programs, promise to save time by detecting memory corruption, input handling, exception cases, and other issues. The goal of this work is to empower developers to prioritize their quality assurance by analyzing the history of bugs generated by OSS-Fuzz. Specifically, we examined what has happened when a project adopts fuzzing as a quality assurance practice by measuring bug lifespans, learning opportunities, and bug types. We analyzed 44,102 reported issues made public by OSS-Fuzz prior to March 12, 2022. We traced the Git commit ranges reported by repeated fuzz testing to the source code repositories to identify how long fuzzing bugs remained in the system, who fixes these bugs, and what types of problems fuzzers historically have found. We identified the bug-contributing commits to estimate when the bug containing code was introduced, and measure the timeline from introduction to detection to fix. We found that bugs detected in OSS-Fuzz have a median lifespan of 324 days, but that bugs, once detected, only remain unaddressed for a median of 2 days. Further, we found that of the 8,099 issues for which a source committing author can be identified, less than half (45.9%) of issues were fixed by the same author that introduced the bug. The results show that fuzzing can be used to makes a positive impact on a project that takes advantage in terms of their ability to address bugs in a time frame conducive to fixing mistakes prior to a product release. However, the rate at which we find authors are not correcting their own errors suggests that not all developers are benefiting from the learning opportunities provided by fuzzing feedback.

9. †*T.H.E.S.E. are the Human Errors Experienced by Software Engineers.* (Under Review). **Benjamin S. Meyers**, Brandon N. Keller, and Andrew Meneely. ACM Transactions on Software Engineering and Methodology *(TOSEM).*

   This study is discussed in detail in Chapter 4.

   Abstract: Software engineers work under strict time constraints, balancing a complex, multi-phase process on top of user support and professional development. Despite their best efforts, software engineers experience **human errors**. Practices such as defect tracking help developers reflect on the outcomes of their errors (*i.e.* software failures), and even the faults that led to those failures, but not the underlying human behaviors. While human error theory from psychology research has been studied and applied to medical, industrial, and aviation accidents, researchers are only beginning to systematically reflect on developers' human errors. Modern software engineering affords massive, rich records of socio-technical, natural language interaction in public venues, such as with open source software development on GitHub. Specifically, these records can provide insight into developers' **self-admitted** human errors, as evidenced by their apologies. Our broad goal is to help software engineers identify their human errors by systematically aggregating their human errors from existing research and software development artifacts. To that end, this work explores two research studies. We conducted a systematic literature review, called Study 1 (SLR), of 68 research studies about human errors in software engineering. Using established terminology and multiple taxonomies from existing psychology research, we systematically examined a subset of 81,007 search results from three research paper databases, and aggregated software engineers' human errors (192 in total) into a Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.). Study 1 (SLR) resulted in a taxonomy with 12 categories of human error spanning slips, lapses, and mistakes from James Reason's popular Generic Error-Modelling System. We found that mistakes (both rule- and knowledge-based) were the most frequent human errors discussed in literature, a differing result from the psychologist Reason's findings for general-purpose human errors. We also found that over half (53%) of studies did not explore specific categories of human error (beyond high-level theory categories) and only 15% of studies had a scope general to all phases of the software development lifecycle. In Study 2 (Mining), we collected 88.6 million developer comments from GitHub. Using apology lemmas from linguistics research, we created an automatic apology classification process. We manually categorized a random subset of 332 apology comments according to the experienced human error. We compared human error categories derived from developers' apologies with the Study 1 (SLR) version of T.H.E.S.E. Our automatic apology classification achieved near perfect recall (99%) with high accuracy (91%). 2.7 million developer comments were classified as apologies. We found that software engineers apologize in 3.15% of all of the comments in our dataset, a frequency of apologies over one thousand times greater than those documented in the Switchboard corpus. We identified 15 categories of human error not previously documented in literature. We present T.H.E.S.E. as a combination of human errors identified in both studies, containing 30 categories (7 slips, 8 lapses, 15 mistakes) of human error. This work reveals that slips, lapses, and mistakes from cognitive psychology can meaningfully describe software engineers' self-admitted human errors. By identifying their human errors, software engineers can better find areas of improvement, while managers can identify systemic human errors across development teams and implement process improvements in response.

10. *Taxonomy-Based Human Error Assessment for Senior Software Engineering Students.* (Forthcoming 2024). **Benjamin S. Meyers** and Andrew Meneely. Special Interest Group on Computer Science Education *(SIGCSE)* Technical Symposium.

    This study is discussed in detail in Chapter 5.

    Abstract: Software engineering is a complex symphony of development activities spanning multiple engineering phases. Despite best efforts, software engineers experience **human errors**. Human error theory from psychology has been studied in the context of software engineering, but human error assessment has yet to be adopted as part of typical post-mortem activities in software engineering. Our goal in this work is to evaluate an existing Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.) as a learning tool for software engineering students. We conducted a user study involving five software engineering students at RIT. In two experimental phases (17 weeks total), participants self-reported 162 human errors that they experienced during software development. Participants' feedback collected via surveys indicates that T.H.E.S.E. is clear, simple to use, and general to all phases of software engineering. Participants also indicated that human error assessment guided by T.H.E.S.E. (1) would benefit other students and professional software engineers and (2) enhanced their understanding of human errors in software engineering and of their own human errors. These are promising results indicating that human error assessment facilitated by T.H.E.S.E. is a valuable learning experience for software engineering students. We release anonymized survey responses and reported human errors. Future work should examine T.H.E.S.E. as a learning tool for professional software developers and examine other software engineering artifacts to identify categories of human error that are not presently captured by T.H.E.S.E.

1.2.2 Data, Source Code, and Tools

In addition to the publications listed in the previous section, this dissertation makes the following contributions:

1. **Data: 88.6 Million Developer Comments from GitHub [214]**

    This is a collection of software developer comments from GitHub issues, commits, and pull requests. We collected 88,640,237 developer comments from 17,378 repositories. In total, this dataset includes: 54,252,380 issue comments (from 13,458,208 issues), 979,642 commit comments (from 49,710,108 commits), and 33,408,215 pull request comments (from 12,680,373 pull requests). Details on data collection and organization can be found in Meyers & Meneely [214]. This dataset was collected in pursuit of the results presented in Chapter 4.2. This dataset will save future researchers a large amount of time and effort, enabling them to focus on data analysis rather than collection.

2. **Data: 1,237 Annotated Developer Apologies from GitHub [215]**

    This is a collection of software developer comments from GitHub with automated apology annotations (as described in Section 4.2.2.2). This dataset also includes manual apology classifications from two annotators and resolutions for their disagreements. Apology mining is a relatively new area of study, so we hope this dataset can serve as a gold standard for future researchers.

3. **Data: 200 Annotated Developer Human Errors from GitHub [216]**

    This is a collection of software developer comments from GitHub with manual human error categorizations (as described in Section 4.2.2.3). Since human error in software engineering is a growing research domain, we hope this dataset will provide value to future researchers.

4. **Data: 162 Human Errors Descriptions [217]**

    This is a collection of human error descriptions with T.H.E.S.E. categorizations and related discussion collected during our user study with software engineering students in Section 5.2. Since human error in software engineering is a growing research domain, we hope this dataset will provide value to future researchers.

5. **Data: Systematic Literature Review of Human Errors in Software Engineering**

   As described in Section 4.1, we conducted a systematic literature review of 68 research studies (identified from 284 total papers) which yielded 192 total human errors. Discussion of these research studies can be found in Section 3.1. This SLR should serve as a concrete foundation for future researchers exploring human error in software engineering.

6. **Tool: Automated Apology Classifier**

   In Section 4.2.2.2, we outline a naïve keyword-based approach to automatically classifying apologies in natural language. This approach accounts for false-positives and yields near perfect recall (99%) and high F1 (87%). Apology mining is a relatively new area of study, so we believe this approach will be beneficial to future researchers.

7. **Tool: Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.)**

   The core contribution of this work is T.H.E.S.E., a taxonomy of 31 categories of human error in software engineering spanning slips, lapses, and mistakes. Details on how this taxonomy was developed can be found in Chapter 4. T.H.E.S.E. is the distillation of 192 human errors from related literature and 200 human errors from developer comments on GitHub. We envision T.H.E.S.E. serving as a shared vocabulary for software engineers to discuss their human errors.

8. **Tool: Human Error Informed Micro Post-Mortem Process**

   In Section 6.1, we outline a human error reflection process that we designed to accompany T.H.E.S.E. This process makes it easy for software engineers to confront and reflect on their human errors, as indicated by our user study participants in Section 5.2.

9. **Tool: Human Error Reflection Engine (H.E.R.E.)**

   In Section 6.2, we describe a semi-automated workflow that facilitates our human error informed micro post-mortem process on GitHub. This workflow serves as a proof-of-concept while also lowering the barrier to entry for software engineering teams who wish to adopt human error reflection with T.H.E.S.E.

10. **Code: Downloading Developer Comments & Apology Classification**[2]

    We release the code used to collect software developer comments from GitHub and classify their apologies. We provide this code in part to ensure reproducibility of our work, but primarily because apology mining is a relatively new area of study, and we hope researchers can build on our novel approach.

11. **Code: Evaluating Sentence-BERT Models for Human Error Classification**[3]

    We release the Jupyter notebooks used to evaluate pretrained Sentence-BERT models for human error classification. We provide this code for reproducibility of our classification experiment and to encourage future researchers to explore their models and data.

[2]`https://github.com/meyersbs/developer-apologies/`
[3]`https://github.com/meyersbs/sbert-these`

Chapter 2

Background

2.1 Human Error

Human errors are actions that lead to unintended, unexpected, or undesirable outcomes [320]. In cognitive psychology, a **human error** is an action that results in something that was "not intended by the actor; not desired by a set of rules or an external observer; or that led the task or system outside its acceptable limits [320]." Human errors are typically classified using one of three theories of human error—Rasmussen's SRK Model [293], Reason's GEMS Framework [298], or errors of omission and commission. A visual summary of these theories is provided in Figure 2.1. All three theories are discussed, with examples, in the remainder of this section.

2.1.1 Rasmussen's SRK Model

Jens Rasmussen classified human behavior as either skill-based, rule-based, or knowledge-based, with each type of human behavior having an associated type of error [293, 295]. Skill-, rule-, and knowledge-based behavior (and errors) exist on a continuum of least-to-most conscious involvement, respectively [87]. These human error types are defined as follows:

- **Skill-Based Errors:** Unconscious errors in highly-practiced/learned automatic sensorimotor tasks, such as starting a car, flipping a light switch, or locking a computer screen. Errors of this type might be putting the wrong key in the ignition, flipping the wrong light switch, or forgetting to lock a computer screen.
- **Rule-Based Errors:** Semi-conscious errors in rule-based behaviors (*i.e.* defined processes), such as following the instructions to build a Lego® set, following a recipe, or installing software dependencies in the correct order. Errors in these examples might include skipping a step when building a Lego® set, mixing ingredients in the wrong order, or trying to install `scikit-learn` before `numpy` has been installed.
- **Knowledge-Based Errors:** Conscious improvisation in unfamiliar situations where no rules or routines exist, such as free-hand climbing a mountain for the first time, navigating in a forest with no marked trails, or driving a car with a manual transmission for the first time. In these examples, errors might be stepping on a loose rock, turning North instead of South, or not shifting gears at the appropriate speeds.

Rasmussen's human error theory, commonly referred to as the Skill-Rule-Knowledge (SRK) model, has been applied to domains such as human-computer interaction [39, 185, 186, 370], aircraft maintenance [130], industrial installation [294], aviation [318], police accountability [337], and medicine [71, 182].

2.1.2 Reason's GEMS Framework

Influenced heavily by the work of Rasmussen [295], James Reason combined two previously distinct areas of human error research—slips and lapses, and mistakes—in his Generic Error-Modelling System (GEMS) [298]. Slips and lapses are "the failure of actions to go as intended [298]", while mistakes are "the failure of intended actions to achieve their desired consequences [298]." Put differently, mistakes are errors that result from an inadequate plan [298, 320]. Formally, slips, lapses, and mistakes are defined as follows, with examples from Anu *et al.* [22]:

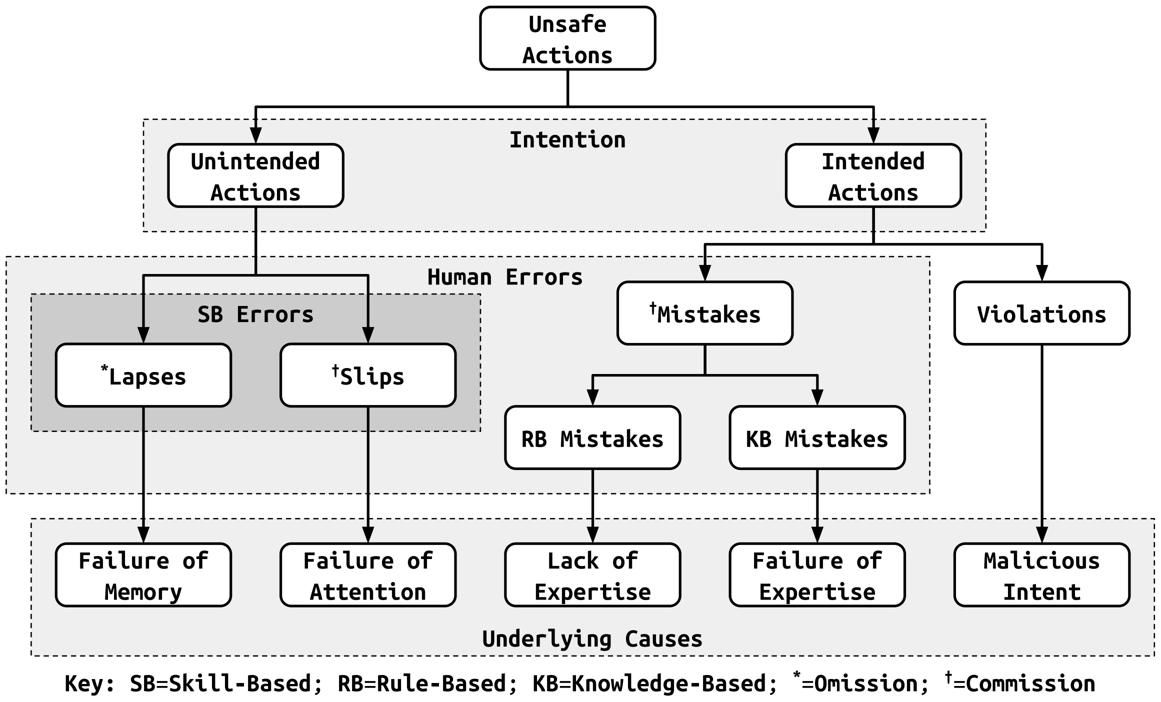


Figure 2.1: Summary of Human Error Theories

James Reason [298] places all human errors and malicious actions (*i.e.* violations) under the umbrella of *unsafe actions*, which can be divided into *unintended actions* (slips and lapses) and *intended actions* (mistakes and violations). Mistakes are further divided into rule-based and skill-based mistakes, while slips and lapses are both skill-based errors. Slips result from a failure of attention; lapses from a failure of memory; rule-based mistakes from a lack of expertise; knowledge-based mistakes from a failure of expertise; and violations from malicious intent.

- **Slips:** Failing to complete a properly planned step due to inattention [261, 320], such as putting the wrong key in the ignition. Another example would be Freudian slips of the tongue [105] where a person misspeaks in a way that could reveal what they are currently thinking.
- **Lapses:** Failing to complete a properly planned step due to memory failure [298, 301, 320], such as forgetting to put the car in reverse before backing up, or forgetting to check if a pointer is non-null before dereferencing it. Lapses likely occur when the load on working memory is high [56].
- **Mistakes:** Planning errors that occur when the plan is inadequate [298, 320], such as getting stuck in traffic because you didn't consider the impact of the bridge closing, or choosing an inadequate sorting algorithm.

There is some overlap with Rasmussen's skill-, rule-, and knowledge-based errors: slips and lapses are both skill-based errors, while mistakes can be rule-based or knowledge-based errors [87, 298, 318]. Reason says knowledge-based mistakes usually result from *failures of expertise*, while rule-based mistakes usually result from a *lack of expertise* [298]. Additionally, Reason mapped slips, lapses, and mistakes to the cognitive stages where these errors occur:

- **Cognitive Planning:** Mistakes originate from failures in the cognitive "processes concerned with identifying a goal and deciding upon the means to achieve it [298]."
- **Cognitive Storage:** Lapses originate from failures in the cognitive storage and retrieval of information related to a plan [298].
- **Cognitive Execution:** Slips originate from cognitive execution failures in the "processes involved in actually implementing the stored plan [298]."

Reason also describes *violations*, which are actions motivated by malicious intent. Violations are outside the scope of this work, since a malicious software engineer (*e.g.* an insider threat) is not likely to admit to their violations publicly. Reason's GEMS framework has mainly been applied in the medical domain [98, 159, 305] and to study automobile accidents [300, 332].

2.1.3 Errors of Omission & Commission

While discussed in psychology and medical research, we could not identify the original source or definitions for errors of omission and commission. Sigmund Freud's discussion of omissions in writing [105] may be the first occurrence of omissions in psychology. Or perhaps errors of omission and commission evolved from the Christian concepts of sins of omission (*i.e.* choosing not to do something that you should do) and sins of commission (*i.e.* performing a reprehensible act), first recorded in English in the 1400s [270, 271]. Aside from the medical domain, errors of omission and commission have made their way into legal definitions [49, 129]. Based on general consensus, we provide the following definitions:

- **Errors of Omission:** Failure to perform a required action, such as choosing not to file your taxes, choosing not to stop at a red light, and storing sensitive data without encryption.
- **Errors of Commission:** Failure to perform an action in an appropriate manner or at the appropriate time, such as rolling stops at a stop sign, turning in an assignment late, or implementing client-side instead of server-side authentication.

Lapses are commonly considered to be errors of omission [298, 320] and slips are considered errors of commission [203, 318]. At least one definition also considers mistakes to be errors of commission [203].

Noteworthy work using errors of omission and commission outside the medical domain includes Swain & Guttman's human reliability analysis focused on nuclear power plants [336]. Swain & Guttman classified errors in individual discrete actions as omissions (*i.e.* leaving something out) or commissions (*i.e.* doing something incorrectly). Commission errors are further divided into sequence errors (*i.e.* doing something out of order), timing errors (*i.e.* doing something too late/early), and qualitative errors (*i.e.* doing too much/too little).

2.2 Natural Language Processing (NLP)

Natural Language Processing (NLP), at the intersection of linguistics and computer science, is the process of analyzing **natural language** (*i.e.* human languages, rather than programming languages). There are too many techniques and applications of NLP to discuss them all here, but some basics and specifics relevant to this work are outlined in this section. Note that this section focuses on NLP applied to written language, but NLP is also used to analyze spoken language.

2.2.1 Preprocessing

The first step in most applications of NLP is to preprocess a **corpus** of natural language text (*i.e.* a collection of related natural language texts, such as all of Shakespeare's plays). Preprocessing involves cleaning up the text or converting it into a specific form before applying advanced NLP techniques. Preprocessing usually involves removing punctuation and/or converting all of the words to lowercase. Some other common preprocessing steps include:

- **Stop Word Removal:** Stop words are words that contribute to grammatical structure, but convey little-to-no meaning, such as prepositions (*e.g.* at, by, of, to), conjunctions (*e.g.* for, and, but, so), and sometimes pronouns (*e.g.* I, my, our, they).
- **Tokenization:** Tokenizing involves splitting strings of natural language into a list of individual words (called tokens).
- **Sentence Splitting:** Strings of natural language are often split into a list of individual sentences.
- **Lemmatization:** Lemmatizing involves converting tokens into **lemmas**, simpler and common base forms. For example, the words *"bug"*, *"bugs"*, and *"bug's"* all have the same lemma, *bug*.
- **Parts-of-Speech Tagging:** Parts-of-Speech (POS) reveal grammatical information about words in context. Some common POS include nouns (*e.g.* bug, vulnerability, attacker), verbs (*e.g.* programming, fixed, updates), adjectives (*e.g.* dangerous, robust, inconsequential), adverbs (*e.g.* maliciously, diligently), determiners (*e.g.* the, some, those), and pronouns.

2.2.2 Word Embeddings

Researchers are frequently interested in the **semantic** content of words—the meaning of words in the context of the words surrounding them. **Word Embeddings (WE)** are "vector-space representations of the semantic meaning of words within a corpus [222]." Simple feed-forward neural networks can be trained to derive embeddings of individual words based on their context, resulting in fixed-length vectors (called **word vectors**) of real numbers. The individual numeric values in a word vector represent an unique dimension of the different meanings associated with a word. For example, the words "vulnerability" and "bug" will have similar word vectors since they are used in similar contexts, whereas the words "hacker" and "kitchen", being used in different contexts, will have dissimilar word vectors.

For an individual document in a corpus, word vectors can be summed to yield a **document vector**. Document vectors can be compared to assess the semantic similarity between two texts using machine learning techniques like clustering [213, 244, 275]. The downside of word and document vectors is that they are not human-interpretable.

2.2.3 BERT, Sentence-BERT, and Cosine Similarity

Bidirectional Encoder Representations from Transformers (BERT) models also output embeddings that represent the meaning of words within a corpus. The key difference between BERT and word embeddings is that BERT models derive meaning from the entire sentence, rather than individual words. BERT models learn meaning bidirectionally with two pre-training phases: (1) Masked Language Modeling (MLM), and (2) Next Sentence Prediction (NSP). In MLM, tokens in a sentence are *masked* (*i.e.* hidden) from the model at random. The model then attempts to predict the masked tokens using the context on either side (bidirectionally) of the masked token. Bidirectional learning enables BERT models to represent semantic context better than traditional language models. BERT models are also more powerful than previous language models because they consider sentence relationships using NSP. In NSP, the model is shown pairs of sentences, with Sentence A preceding Sentence B. 50% of the time, Sentence B is the actual sentence that follows Sentence A, and 50% of the time, Sentence B is a random sentence from the corpus [77].

Sentence-BERT (or S-BERT) is an expansion of BERT with an extra pooling function that results in fixed-length sentence embeddings. This allows sentence embeddings from Sentence-BERT models to be compared using **cosine similarity**—the cosine of the angle between two vectors (*e.g.* sentence embeddings); vectors with similar direction (*i.e.* representing similar semantic content) will have high cosine similarity [250]. Both BERT and Sentence-BERT models can be fine-tuned for specific tasks using labeled data. Fine-tuning BERT and Sentence-BERT models is beyond the scope of this dissertation, so we will not discuss it further, but details can be found in Devlin *et al.* [77] and Reimers & Gurevych [303].

In this work, we make use of various pre-trained Sentence-BERT models using the `sentence-transformers` package (version 2.2.2) from HuggingFace [1].

2.2.4 Linguistic Characteristics

Linguistic characteristics discussed in Section 1.2.1 (my prior work) are defined below, along with a brief description of how they were collected/computed. While the main contributions of this dissertation do not make use of these linguistic characteristics, we include them here as they were a key part of our journey.

- **Inquisitiveness:** The inquisitiveness metric (from [245]) is an attempt to quantify speculative types of conversations in code review. This naïve metric is simply the count of question marks (`?`) in text.

- **Sentiment:** We computed sentiment—a measure of how positive or negative the tone of natural language is—using the model in Stanford CoreNLP [197], which uses information about words and their relationships to assign a sentiment value (on a 5-point scale from very negative to very positive) to a natural language text. In our prior work [218, 245], we collapsed "very positive" and "positive" into a single "positive" label, and did the same for "negative" labels.

- **Syntactic Complexity:** Early metrics aimed at measuring syntactic complexity—how complex a sentence's structure is—focused mainly on sentence length (number of words) [100]. We used two metrics, Yngve [368] and Frazier [102, 103, 104], which assign a complexity score to sentences based on structural information extracted from syntactic parse trees. The algorithms for computing these scores are discussed in prior work [245]. The key distinction is that Yngve measures the breadth of the syntactic parse tree, while Frazier measures the depth.

[1] `https://huggingface.co/sentence-transformers`

- **Information Content:** In our prior work [218], we computed two measures of information content. The first, content density, is a ratio of open-class (*i.e.* POS types that commonly accept new words, such as nouns and verbs) to closed-class (*i.e.* POS types that rarely accept new words, such as prepositions) words in a natural language text. The second, propositional density, is the ratio of propositions to the number of words in a text [310]. Propositions were identified using a similar approach to Brown [52].
- **Politeness:** Danescu-Niculescu-Mizil *et al.* [70] collected 10,000 natural language utterances from Wikipedia and Stack Exchange conversations, and used Amazon Mechanical Turk to crowdsource politeness annotation. Each utterance was assigned a label of "very polite", "polite", "neutral", "impolite", or "very impolite" by five annotators. The final politeness score for each utterance was the average of the five individual scores. These annotations were used to train a classifier which assigns a label of "polite" or "impolite" to new utterances. In our previous work [219], we used the classifier from Danescu-Niculescu-Mizil *et al.* [70] to compute politeness scores.
- **Uncertainty:** Uncertainty in natural language manifests itself as a lack of information required to confirm whether a proposition is true [348]. There are four types of uncertainty—epistemic, doxastic, investigative, and conditional—with definitions and examples provided in our prior work [219]. We used the corpus from Farkas *et al.* [97] and the feature set described by Vincze [348] to train a multi-label regression model capable of classifying each uncertainty type [218].
- **Formality:** Lahiri [176] annotated a corpus of 7,032 sentences on a numerical scale from 1 to 7 for three metrics: formality, informativeness, and implicature, all rooted in Grice's pragmatic theory [122]. Formality is a measure of the observance of etiquette taken in forming a sentence [128, 176]. We reduced the 7-point scale to a binary scale and trained a formality classifier on Lahiri's SQUINKY corpus using the same model algorithm and features used in uncertainty classification. Our classifier outputs a numerical value between 0 and 1 indicating low to high formality, respectively [218, 219].
- **Informativeness:** The informativeness metric measures the ability of language to lead to mutual agreement. Informative language should be clear, direct, and unambiguous [128, 176]. We used the same classification method described for formality to compute informativeness scores in our prior work [219].
- **Implicature:** Implicature measures how much context is missing from natural language [128]. We computed implicature using the same classification method described for formality [219].

2.3 Software Engineering Concepts

2.3.1 Software Development Lifecycle

There are a variety of models used to organize software development processes, each with their own benefits and drawbacks. Rather than discussing them all here, we discuss the six main processes involved in these models:

- **Requirements & Planning:** Software must meet the needs of the stakeholder(s) (*i.e.* customers). During the requirements & planning process, software engineers elicit requirements from the stakeholder(s) by asking questions about their goals for the software, what it should and should not do, the scope of its usage, *etc.* The information gained in this phase of software development is essential to designing, implementing, and deploying the best (as defined by the requirements) software product.
- **Design:** During the design phase, software engineers translate the stakeholders' requirements into "a representation of the software that can be assessed for quality before coding begins [288]." The design serves as blueprints for the implementation phase.
- **Implementation:** During implementation, the design for the software system is translated into source code. The majority of coding occurs during this phase. Code reviews primarily take place during implementation, but also during maintenance.
- **Testing:** The testing phase involves designing and writing tests (*e.g.* unit tests, integration tests), running those tests, and evaluating the state of the system based on the outcome of the tests. If tests fail, the source code is modified to ensure that the tests pass.
- **Deployment:** During deployment, the software system (in a stable form) is installed in a production environment that the stakeholders and their users will access. Configuration for the software itself occurs during deployment, as does configuration of any software dependencies.

- **Maintenance:** Despite best efforts to produce robust software, bugs and vulnerabilities inevitably fall through the cracks. During maintenance, fixes/patches for bugs and vulnerabilities are implemented, tested, and deployed to the production environment.

2.3.2 Version Control, Code Review, and Pull Requests

Version control systems allow development teams (or individual developers) to document changes to the source code stored in a **repository**. Version control tools also allow (1) changes by multiple developers to the same code to be merged with conflict resolution processes, (2) changes to be reverted, and (3) separate branches (*i.e.* copies of the main source code) to facilitate implementation of new features and/or distinct versions of the software [48]. Modern examples of version control systems are Git and Subversion.

The process of examining proposed changes to source code is called **code review**. Code review aims to improve the overall quality of software systems through the following activities: identifying changes to source code that need to be reviewed, determining and prioritizing which aspects of the changed source code need to be reviewed, reviewing the changed code, providing feedback to the author of the changed code, and implementing feedback from the code review. Code reviews identify potential bugs and vulnerabilities in software and afford developers the opportunity to fix those errors before they can impact the production system and/or the software's users.

Historical code reviews occurred in person, but much of modern software development occurs remotely, with developers located all over the world. **Pull requests** now serve as a form of code review, allowing (1) developers to notify each other that they have proposed changes to the source code that require review, (2) other developers to provide feedback on the changed code, and (3) implementation of that feedback prior to merging the accepted changes into the primary version control branch.

2.3.3 Open vs. Closed Source Software

Software's source code can either be **open**—publicly visible—or **closed**—visible only to authorized people. Further characteristics of open source software are defined by the Open Source Initiative [272], but they do not need to be discussed here. Closed source software projects are usually only accessible to employees at a specific company working on a specific project. For example, Microsoft[2] has many closed source projects, including Skype[3], 365 (previously "Office")[4], and the Windows OS[5]. Conversely, open source software projects often have a mix of company employees and non-affiliated volunteer developers, such as the Ubuntu OS[6], which has developers from Canonical[7] and volunteer developers.

2.3.4 Git & GitHub

Git is an open source version control system and supply-chain management tool designed to handle any size project with speed and efficiency, which is used by major software companies like Google, Microsoft, and the Linux kernel [55]. Git stands apart from other version control systems due to the flexibility of its branching functionality. Developers can create any number of branches, allowing them to quickly test ideas in separate branches and delete or keep those branches with ease. Branches can also contain separate versions of the source code for different platforms, or just stable and experimental points in the development lifecycle [54].

GitHub is a website that hosts git repositories for users and organizations, which include popular companies—such as Netflix, AirBnB, Reddit, Shopify, and Lyft [330]—and government organizations [111]. Users and organizations can create, update, and delete both public and private repositories. GitHub also facilitates bug/issue tracking, pull requests, and continuous integration for repositories. At the time of writing, GitHub has over 200 million repositories, spanning over 65 million developers and over three million organizations [110].

A **GitHub bot** is a program that automatically interacts with GitHub repositories. Common GitHub bots include Dependabot[8], which monitors repositories for out-of-date dependencies and submits pull requests updating those dependencies, and probot-stale[9], which closes issues and pull requests that have been inactive for too long.

[2]`https://www.microsoft.com/`
[3]`https://www.skype.com/`
[4]`https://www.microsoft.com/en-us/microsoft-365`
[5]`https://www.microsoft.com/en-us/windows/default.aspx`
[6]`https://ubuntu.com/`
[7]`https://canonical.com/`
[8]`https://dependabot.com/`
[9]`https://github.com/probot/stale`

GitHub Actions allow for automated software workflows on GitHub, such as building, testing, and deploying code, and issue tracking management [114]. GitHub actions can also leverage **docker containers**—sandboxed, standalone software applications packaged with their dependencies and configurations [82].

2.4 Computer Security Concepts

2.4.1 Bugs, Vulnerabilities, and Defects

McGraw defines a **bug** as "an implementation-level software problem [204]." Bugs manifest in unexpected system behavior. A **vulnerability** is a bug or a *design flaw* with security consequences—vulnerabilities violate the confidentiality, integrity, or availability of a piece of software. Bugs (*e.g.* text displaying with the unintended font) and vulnerabilities (*e.g.* lack of authentication) are **defects** that may lie dormant in software for years before being noticed, or they may never be observed at all.

2.4.2 Exploits & Threats

An **exploit** is a piece of software, a chunk of data, or a sequence of commands that takes advantage of a vulnerability to cause unintended or unexpected behavior. Exploits can be manual (*i.e.* an attacker typing commands) or automated (*i.e.* malware). A **threat** is either a *malicious actor* (*i.e.* hacker or attacker) capable of violating the confidentiality, integrity, and/or availability of software, or a class of exploits (*e.g.* spoofing) [208].

2.4.3 Errors, Faults, and Failures

In computing, an **error** is "a human action that produces an incorrect result [145]." **Faults** are the manifestation of errors in software (*i.e.* the *cause* of a malfunction) which lead to **failures**, an undesired effect observed in software (*i.e.* the malfunction) [48, 145]. For example, a common failure is a segmentation fault due to array boundaries not being checked (fault). The actual error leading to this fault could be that the developer didn't know they needed to check array boundaries, the developer improperly implemented array boundary checking, or perhaps some other human error. Note that the term *defect* is sometimes used to refer to either a fault or a failure when the distinction is unimportant [48].

2.4.4 CIA: Confidentiality, Integrity, and Availability

During vulnerability assessment (*e.g.* CVSS assessment [207]), software engineers and security practitioners are typically interested in the impact of the vulnerability on the confidentiality, integrity, and availability (CIA) of the software system. We summarize these terms briefly here:

- **Confidentiality** is violated when the system discloses information that was intended to be kept secret/hidden to unauthorized parties [101, 267]. For example, credit card numbers transferred over the internet without encryption, or access to medical records by unauthorized family members.
- **Integrity** is violated when the system can no longer be trusted [101, 267]. For example, not being able to trust whether database information is accurate, not being able to trust the source of information, or executing malicious code.
- **Availability** is violated when the system becomes (completely or partially) inaccessible to the user, or so slow that it is effectively inaccessible [101, 267]. For example, the Google Maps application not being able to communicate with GPS satellites, resulting in users getting lost.

2.5 Psychology Concepts

Related work in Chapter 3 references a handful of psychology concepts, which we summarize in this section.

2.5.1 Working Memory

Working memory refers to a system of the human brain that "is responsible for keeping track of multiple task-related goals and subgoals, or integrating multiple sources of information [242]." In the theory of working memory, information can only be held for a temporary period of time and the amount of information that can be held is limited. *Short-term memory* is a common synonym for working memory in colloquial speech, but working memory is considered a separate system from short-term memory, the distinction being that items in working memory can be modified, whereas short-term memory is just a storage mechanism [28]. Miller *et al.* [224] coined the term "working memory" in the context of the computational theory of mind (*i.e.* the human mind is an information processing system, much like a computer). It is commonly assumed that working memory has a limit of seven plus-or-minus two items [223].

2.5.2 Bounded Rationality

When humans make decisions, their rationality is bounded (*i.e.* limited) by factors such as the difficulty of the problem at hand, cognitive capability, and available time. Due to these limitations, humans often choose an adequate solution to a problem, rather than an optimal solution based on a full cost-benefit analysis [57]. Herbert Simon proposed the idea of **bounded rationality** [327, 328] as an alternative model of decision-making to the classical model, which assumes that all decision-making is purely rational [108].

2.6 Taxonomies

Broadly, a taxonomy is a system of classification. Among the most well-known taxonomies are the Dewey Decimal System for organizing books in a library [78] and the phylogenetic tree of life, which describes the evolution and relationships between organisms, popularized by Charles Darwin [72]. Many taxonomies have been proposed in the field of security (see Section 3.3). A well-defined security taxonomy, according to Lough [192] and others before him [9, 44, 136, 173, 188], should be generally accepted by the target community, easily understood, objective, specific, and unambiguous. Additionally, the categories in a taxonomy should be mutually exclusive, and classification should be repeatable. Further discussion of these properties of a well-defined taxonomy is provided in Section 4.1.4.

2.7 Statistical Concepts

In Section 4.2 and Section 5.3 we use standard statistical measures of model performance, briefly described here:

- **True Positives (TP)** and **True Negatives (TN)** are correct predictions, when a model correctly classifies a data point as the target-class or not-the-target-class, respectively [361].
- **False Positives (FP)** are incorrect classifications when the model classifies a data point that is not-the-target-class as the target-class. **False Negatives (FN)** are the opposite of FPs, when the model incorrectly classifies a data point that is the target-class as not-the-target-class [361].
- **Precision** is the proportion of true positives to the sum of true positives and false positives, *i.e.* $TP/(TP + FP)$ [361]. For example, when classifying a software developers' error as a slip with a binary model, classifications will either be *slip* or *not a slip*—precision is then the proportion of correctly classified slips to all entries (correctly and incorrectly) classified as slips.
- **Recall** is the proportion of true negatives to the sum of true positives and false negatives, *i.e.* $TP/(TP + FN)$ [361]. Using the same example, recall is the proportion of correctly classified slips to the total number of entries that should have been classified as slips.
- **F-measure** is the proportion of $(2 \times TP)/(2 \times TP + FP + FN)$. F-measure (or F1 score) is the harmonic mean of precision and recall. In many classification problems, high scores for precision and recall are desired, but comparing two or more models using two metrics is difficult; F1 provides a combined form of precision and recall that can be used to compare models when both precision and recall are important [361].

Chapter 3

Related Work

3.1 Human Errors in Software Engineering

This section outlines research related to human errors in software engineering and general computing.

3.1.1 Skill-, Rule-, and Knowledge-Based Errors

In 1987, Rizzo *et al.* [309] gave 16 human subjects with minimal computer experience two hours of training on how to use the Appleworks database system. Then, subjects were split into two groups to perform separate experiments. In the first experiment, subjects were asked to (1) identify if a specific item was present in the database, (2) find three items in the database and change their values, (3) create a new file using items from the database, and (4) create three new files with specific database items and print them out. In the second experiment, participants were tasked with finding an item in the database, modifying its value, and printing it out; this was completed four times with different target items. In both experiments, participants were recorded while thinking aloud. Then, errors were classified as slips, rule-based, or knowledge-based errors. In the first experiment, participants detected and corrected 82-86% of their own errors, with little variation between error type and rate of detection. Results of the second experiment showed that participants detected 94% and 95% of rule-based errors and slips, respectively, but only 68% of knowledge-based errors.

Zapf *et al.* [370] observed 198 (out of 259) office workers—from 15 departments spanning public/private companies and small firms—using office software. Questionnaire data was also collected from 232 office workers. Office workers were asked about their level of computer expertise, how they tend to work, how they react to errors, *etc.* In total, 1,306 distinct errors were observed and collected into a taxonomy, inspired by Rasmussen's [294, 297], Norman's [261], and Reason's [297, 298] work with human errors. The taxonomy (summarized in Table 3.1) had four major error types: inefficiency (*e.g.* bad habits), usability (*e.g.* sensorimotor and knowledge-based errors), functionality (*e.g.* repetitive actions, interruptions), and interaction errors. About one third of observed errors were knowledge-based or other cognitive errors. This is perhaps the first work using Rasmussen's SRK model to understand human error in a true human–computer interaction setting.

In 1994, Cockram *et al.* [64] conducted a survey of past, ongoing, and new software projects. Details of the survey questions, survey respondents, and final results are not available. However, Cockram *et al.* did report two skill-based, two rule-based, and nine knowledge-based errors from one case study. Specific categories beyond skill-, rule-, and knowledge-based errors were not discussed.

In his 2003 doctoral dissertation, David Trepess [345] studied human error in collaborative systems with the goal of improved understanding. Trepess created a classification model for human error in collaborative systems. In this model, human errors can occur at each of three contextual levels—local interactions (*i.e.* tools and users produce tasks), situation context (*i.e.* opportunities and interests produce plans), and social context (*i.e.* structure and action produce goals). Failures in local interactions are classified under Rasmussen's SRK model (see Table 3.2), with the addition of technical failures (resulting from human errors in the social and situation contexts). Human errors in the situation context are classified as conflicts between opportunities, interests, and plans. Similarly, human errors in the social context are classified as conflicts between structures, actions, and goals.

Im & Baskerville [147] replicated a 1993 study by Baskerville [31]—related to information technology hazards (*e.g.* physical assault, malicious code, cracking, programmer error)—and found that, contrary to popular perception (in 2005), human error remained a significant contributor to security issues. In response, Im & Baskerville expanded

Table 3.1: Human Errors Identified by Zapf *et al.* [370]

Category	**Human Error**	**SRK Category**
Inefficiency	Habit	Rule
Inefficiency	Knowledge	Knowledge
Usability	Knowledge	Knowledge
Usability	Sensorimotor	Skill
Usability	Intellectual Regulation: Thought	Knowledge
Usability	Intellectual Regulation: Memory	Knowledge
Usability	Intellectual Regulation: Judgement	Knowledge
Usability	Flexible Action Patterns: Habit	Skill
Usability	Flexible Action Patterns: Omission	Rule
Usability	Flexible Action Patterns: Recognition	Skill
Functionality	Action Blockades	N/A
Functionality	Action Repetitions	N/A
Functionality	Action Interruptions	N/A
Functionality	Action Detours	N/A
Interaction	Interaction	N/A

on the 1993 threat taxonomy by further classifying accidental threats as skill-, rule-, and knowledge-based errors. Im & Baskerville found that the percentage of skill- and knowledge-based errors had increased since the original study, while the percentage of rule-based errors had decreased. Im & Baskerville concluded with a plea for security practitioners to focus on safeguards (*e.g.* training, work aids) to protect information systems from human error.

In the same year, Ko & Myers [168] identified six distinct actions that occur during programming activities (creating, reusing, modifying, designing, exploring, and understanding of requirements, design, code, documentation, *etc.*) and suggested three types of cognitive breakdowns that occur during these activities—skill, rule, and knowledge breakdowns based on Rasmussen's SRK model[1]. Cognitive breakdowns are further divided into two subcategories—skill breakdowns due to inattention or over-attention, rule breakdowns due to using a wrong rule (applicable to most contexts, but not all) or a bad rule (*i.e.* a rule with problematic conditions/actions), and knowledge breakdowns due to bounded rationality (*i.e.* not exploring the entire problem space because working memory is limited) or faulty (*i.e.* incomplete or inaccurate) models of the problem space. Each of these six subcategories is further divided into specific human errors. Developers were recorded thinking aloud in a series of experiments using the Alice 3D programming system. Starting from an observed software error, runtime fault/failure, or cognitive breakdown, Ko & Myers identified the cause and documented chains of causes (*i.e.* chains of cognitive breakdowns) until no further cause could be identified. Ko & Myers observed 102 software errors stemming from 159 cognitive breakdowns. The average breakdown chain consisted of 2.3 breakdowns. The notable contribution of Ko & Myers' work is the idea that chains of cognitive breakdowns between specification, implementation, and runtime activities can compound, *i.e.* errors in software can be the result of multiple human errors at different stages of software development.

In 2008, Keller *et al.* [161] presented ConfErr, a tool that generates realistic software configuration errors, injects them, and measures their effects to assess the resilience of software systems to human errors in configuration. ConfErr generates configuration errors representative of skill-, rule-, and knowledge-based human errors, including spelling mistakes, incorrect structure in a configuration file, repeated/missing configuration options, and configuration values that do not meet the constraints. Through experiments, Keller *et al.* found that MySQL and PostgreSQL often (83% and 76%, respectively) detected spelling mistakes in configurations at startup, while Apache typically (57% of the time) ignored misspellings. Testing unexpected ordering of configuration sections/options, case sensitivity, redundant whitespace, and other structural errors in configuration files revealed that MySQL could not handle mixed-case configuration options, while PostgreSQL and Apache could not handle truncated option names. Finally, Keller *et al.* tested common DNS configuration errors in BIND and djbdns and found that both systems have poor resilience to

[1]However, Ko & Myers only cite Reason's [298] discussion of the SRK model, not Rasmussen's original work [293].

Table 3.2: Human Errors Identified by Trepess [345]

Category	**Human Error**	**SRK Category**
Interaction: Tool	Misrepresentation of Tool	Skill
Interaction: Tool	Inappropriate Tool Selected	Rule
Interaction: Tool	Lack of Knowledge of Tool	Knowledge
Interaction: Tool	Technical Breakdown	N/A
Interaction: Users	Misrepresentation of Users	Skill
Interaction: Users	Inappropriate User Selected	Rule
Interaction: Users	Lack of Knowledge of Users	Knowledge
Interaction: Task	Misrepresentation of Task	Skill
Interaction: Task	Inappropriate Task Selected	Rule
Interaction: Task	Lack of Knowledge of Task	Knowledge
Social Context	Conflicting Structures (Social Norms)	Knowledge
Social Context	Conflicting Histories (Policies)	N/A
Social Context	Conflicting Goals	Knowledge
Situation Context	Conflicting Opportunities	N/A
Situation Context	Conflicting Interests	Knowledge
Situation Context	Conflicting Plans	Knowledge

configuration errors.

Yanyan *et al.* [366] investigated knowledge-based errors by interviewing software developers, but their methodology is unclear. Their findings—that knowledge-based errors can involve incomplete/wrong knowledge or using correct knowledge inappropriately—suggest that they were misidentifying rule-based errors (using correct knowledge inappropriately) as knowledge-based errors. The study identified incomplete domain knowledge and general planning failures as human errors in software engineering.

Huang *et al.* [143] developed a Taxonomy System of Software Developers (TSSD), which initially grouped human errors into four broad categories: cognitive errors, communication failures, procedure violations, and tool misuse. The first version of TSSD was evaluated through a series of interviews with three senior test engineers, three senior software development engineers, and one human error researcher. Feedback from interviews was used to revise TSSD into the version shown in Table 3.3. Note that the human error category column was not provided by Huang *et al.* In addition to the human errors outlined in Table 3.3, Huang *et al.* also identified process errors, tool problems, and task problems as potential software engineering problems outside of human error.

In 2016, Huang *et al.* [142] examined post-completion errors (*e.g.* forgetting to remove your debit card from the ATM, forgetting to remove the original document from a photocopier) in software engineering. Huang *et al.* designed a programming task to (1) calculate multiple nesting levels of a Chinese word, and (2) print a blank line between the results of each nesting level. The study had 55 participants, all undergraduate computer science or software engineering students. Results indicate that participants experienced post-completion errors (forgetting to print a blank line between the results of each nesting level) significantly more often than other errors (*e.g.* code logic errors, syntax errors [144]).

In 2020, Nagaria & Hall [253] conducted semi-structured interviews with 27 professional software developers to investigate eight types of skill-based errors: omitting necessary steps, repeating already completed steps, reversal of actions due to inattention, "unintentionally activating a strongly related action pattern [253]" (*e.g.* if you intended to remove your shoes, but also remove your socks), delays between intention and execution, inattention during repeated tasks, multiple plans or portions of a single plan becoming unintentionally entangled, and general memory lapses. Interviewees identified complexity of development environments as the most common reason for human errors, and lack of concentration (attentional failures) as the most prevalent human error. Interviewees indicated that they struggle to identify effective mitigation strategies for their human errors, "reporting strategies largely based on improving their own willpower to concentrate [253]." Nagaria & Hall conclude with the suggestion that software developers adopt

Table 3.3: Human Errors Identified by Huang *et al.* [143]

Category	**Human Error**	**H.E. Category**
Knowledge Shortage	Insufficient Domain Knowledge	Knowledge
Knowledge Shortage	Insufficient Programming Knowledge	Knowledge
Knowledge Shortage	Insufficient Strategy Knowledge	Knowledge
Knowledge Shortage	Insufficient Linguistic Knowledge	Knowledge
Schema Mismatching	Assuming Problem is the Same as Similar Problems	Knowledge
Working Memory Overload	Problem Too Complicated to Handle	Knowledge
Evaluation Error	Overconfidence	Knowledge
Evaluation Error	Confirmation Bias	Knowledge
Problem Representation Error	Misunderstanding the Problem	Knowledge
Inattention	Interruptions	Slip
Inattention	Perceptual Confusions	Slip
Inattention	Reduced Attention Over Time	Slip
Inattention	Multitasking	Slip
Inattention	Memory Failures	Lapse
Communication Problems	Expression Error	*
Communication Problems	Comprehension Error	Knowledge
Communication Problems	Unidentified Mutual Error	*
Procedure Violations	Intentional Violation	N/A
Procedure Violations	Insufficient Procedure Knowledge	Knowledge
Procedure Violations	Inattention-Based Procedure Violations	Slip
Tool Misuse	Insufficient Tool Knowledge	Knowledge
Tool Misuse	Operation Lapses	Slip
Other Human Errors	N/A	N/A

* Not enough detail/discussion to categorize

James Reason's *swiss cheese model* [299] to reduce human errors by providing multiple layers of tool, process, and management mitigations to software engineering processes. Nagaria also summarized this study in their doctoral dissertation [251].

Other applications of Rasmussen's SRK model include designing smart home user interfaces [372], research pertaining to fault diversity [144], and autonomous system design [67, 68].

3.1.2 Slips, Lapses, and Mistakes

While it does not directly utilize Reason's GEMS framework, the work of Walia & Carver [351, 352] is discussed here to provide background for Anu's work [12, 15, 18, 22, 23]. In 2009, Walia & Carver performed a systematic

Table 3.4: Human Error Taxonomy (HET) from Anu *et al.* [15]

Human Error	**GEMS Category**
Clerical Errors	Slip
Lack of Consistency in Requirements Specification	Slip
Loss of Information from Stakeholders	Lapse
Accidentally Overlooking Requirements	Lapse
Application Errors	Mistake
Environment Errors	Mistake
Information Management Errors	Mistake
Wrong Assumptions	Mistake
Poor Understanding of Each Other's Roles	Mistake
Mistaken Belief that it is Impossible to Specify Non-Functional Requirements in a Verifiable Form	Mistake
Not Having a Clear Demarcation Between Client and Users	Mistake
Lack of Awareness of Sources of Requirements	Mistake
Problem-Solution Errors	Mistake
Inadequate Requirements Process	Mistake
Syntactic Errors	Mistake

literature review of 149 papers spanning software engineering, psychology, and human cognition domains to identify sources (human errors) for software requirements faults. Walia identified 14 requirements error classes from software engineering and human cognition literature and placed them into the Requirement Error Taxonomy (RET) with three major requirements error types: (1) people (*i.e.* communication, participation, domain knowledge, specific application knowledge, process execution, and other cognition errors), (2) process (*i.e.* management and requirement elicitation/analysis/traceability errors, in addition to errors related to having inadequate methods of achieving goals/objectives), and (3) documentation (*i.e.* organization and specification errors, in addition to having no standard for documenting errors). The RET served as a building block for Anu's Human Error Taxonomy (HET), which further organized human errors in the software requirements phase into Reason's slips, lapses, and mistakes.

Starting in 2016, Vaibhav Anu's doctoral dissertation work presented, refined, and evaluated the HET to help aid developers in early fault detection with the goal of improving overall software quality [12, 15, 18, 22, 23]. Human error classes which occur during the requirements phase (identified by Walia [352]) were mapped to slips, lapses, and mistakes to create the HET (recreated in Table 3.4). Slips include clerical errors and lack of consistency in requirements specifications. Lapses include loss of information from stakeholders and accidentally overlooking requirements. Mistakes include wrong assumptions, poor understanding of team member roles, lack of clear distinction between clients and users, lack of awareness of requirements sources, inadequate requirements process, "mistaken belief that it is impossible to specify non-functional requirements in a verifiable form [15]", and various error categories (application, environment, information management, syntactic, problem-solution) [12, 15, 18, 22, 23].

In [15], Anu *et al.* found that slips and lapses together make up 65% of requirements errors, which is inline with Reason's findings that 61% of human errors are slips and lapses [298]. In experiments, Anu *et al.* found that using HET improved fault detection effectiveness by 225% compared to just using fault checklists. Experimental subjects gave positive reviews of the HET, indicating that it was simple/intuitive, easy to understand/use, and comprehensive [15]. This feedback is consistent with similar HET studies by Hu *et al.* [138, 139].

While the majority of Anu's work focused on human error during the requirements phase [12, 13, 15, 17, 18, 21, 22, 23], his other work has studied the incorporation of human error in software engineering education [16, 19] and programmer-induced vulnerabilities [20].

In 2017, Anu *et al.* [16] conducted an experiment where 16 graduate students in a *Software Requirements Definition and Analysis* course inspected software requirements documents for faults. Then—after receiving training

Table 3.5: Developer Human Error Taxonomy (Dev-HET) from Anu *et al.* [20]

Human Error	**GEMS Category**
Overlooking the Design Documentation	Slip
Overlooking or Failure to Read the API Documentation	Slip
Forgetting to Remove Debug Log Files when Software is Transitioned from a Debug State to Production	Lapse
Forgetting to Remove Test Code Before Deploying Their Applications	Lapse
Forgetting to Fix Those Issues that are Bookmarked in Earlier System Versions (*i.e.* Bookmarked to be Fixed in Later Versions)	Lapse
Forgetting to Fix "Self-Injected" Backdoors in the System	Lapse
Forgetting to Check Every Access to Every Object Because Security Relevant Code is Distributed Between Functional Code	Lapse
Bounded Rationality When Choosing Libraries	Mistake
Incorrect Assumptions or Lack of Knowledge About the Type of Environment in Which His Program Would be Running	Mistake
Lack of Knowledge About Handling Exceptional Conditions	Mistake
Wrong Assumptions About the Potential Program Inputs	Mistake
Wrong Assumptions About User Authorization	Mistake
Blind Trust on Code from Reputable Sources (*e.g.* API Code)	Mistake
Incorrect Assumption that Blindly Following the Specifications Generated During the Design Stage of SDLC Guarantees Security	Mistake
Incorrect Assumption that Developers Should Only Perform Functional Testing and Security Testing is Testing Team's Responsibility	Mistake

on Reason's slips, lapses, and mistakes, and how to use HET—these students re-inspected their assigned software requirements documents using human errors to identify faults. Students found an average of six faults in the initial inspection and an average of 14 new faults in the re-inspection using HET to identify faults—a 233% increase in fault detection effectiveness. A second experiment with a different requirements document and 34 different graduate students was conducted, and post-inspection survey responses were collected. Survey results indicated that students could confidently distinguish between slips, lapses, and mistakes and saw the benefit of using HET during error-based inspections of software requirements documents.

Two years later, Anu *et al.* [19] again evaluated the effectiveness of using HET to improve fault detection in software requirements document inspections in two experiments—one with industry professionals and another with undergraduate students. In the first experiment, 11 employees at PowerObjects (an organization focused on creating high-quality requirements for their clients) were given training on HET and a Human Error Abstraction Assist (HEAA) tool [14] (intended to aid inspectors in human error identification). Then the participants were shown 15 faults in a requirements document and asked to identify (1) at what stage of requirements engineering (analysis, specification, elicitation, or management) the human error was made, (2) whether the human error was a slip, lapse, or mistake, and (3) what specific error from the HEAA was made. Participants also answered questions about whether they had encountered this type of error before, what extra information would have helped them understand this type of error, and how they would prevent that error in the future. Results indicated that the majority of participants believed they had committed or seen the human errors from HET in previous software projects. In the second experiment, 36 undergraduate students in a *Principles of Software Engineering* course worked in teams to create their own software requirements specification documents. Students were given similar training on HET and HEAA as the industry professionals and tasked with performing a traditional fault checklist-based inspection and a re-inspection informed by HET/HEAA. Similar to the findings of [16], students were able to identify more faults in requirements specifications using HET/HEAA than traditional fault-checklists [19].

In 2020, Anu *et al.* [20] examined the common human errors committed by software developers that lead to vulnerabilities in software. From an analysis of 104 vulnerabilities from the National Vulnerability Database (NVD) and a literature survey, Anu *et al.* produced the Developer Human Error Taxonomy (Dev-HET), which contains two human error classes considered slips, five considered lapses, and eight considered mistakes (recreated in Table 3.5). Slips include overlooking design documentation and overlooking/failing to read API documentation before using the API. Lapses include forgetting to remove debug log files and test code before deploying software systems, forgetting to fix previously identified issues, forgetting to remove intentional back doors into the system, and forgetting to check every access to every object in the source code. Finally, mistakes include bounded rationality when choosing libraries, incorrect assumptions/lack of knowledge about the production environment, wrong assumptions about potential inputs and user authorization, lack of knowledge about exception handling and exception conditions, blind trust in third-party code, and the incorrect assumptions that (1) blindly following design guarantees security and (2) security testing is the responsibility of the testing team, not the implementation team.

Aside from significant contributions by Anu *et al.*, other computing research utilizing slips, lapses, and mistakes includes Ahmed's brief literature review of human factors/errors in information security [3], Kraemer & Carayon's conceptual framework intended to identify the organizational factors that lead to human errors, which can lead to computer and information security vulnerabilities [171], and Nagaria & Hall's work in reducing software developers' human errors by improving situational awareness [252].

Kraemer & Carayon [171] interviewed eight network administrators from two academic computer laboratories and eight information security specialists from five domains (retail, finance, energy, health care, and manufacturing) about errors that contribute to security breaches and factors that precipitate these errors. Interview responses indicated that vulnerabilities are often the result of unintentional errors (*i.e.* mistakes) made by network administrators and intentional errors (*i.e.* violations) made by end users. Notable factors contributing to human error identified in interview responses include disparities among individual network administrators' security priorities (*i.e.* subjectivity regarding the importance of various security practices), end users' misunderstanding or lack of security knowledge, keeping up with large workloads/backlogs, workplace environment (*e.g.* noise and lighting), lack of adequate hardware/software to maintain proper security, and poor communication between employees at different organizational levels. Kraemer & Carayon identified Reason's GEMS framework as being consistent with how network administrators and security specialists view human errors.

Borrowing from other domains (*e.g.* medicine and transportation), Nagaria & Hall [252] sought to reduce human error by improving situational awareness, the "understanding of what is going on around you while performing a task [252]" (*e.g.* perceiving the environment, comprehending the current situation, and predicting the future situation). Nagaria & Hall asked 10 software developers to record their errors over a five-day period, then provided OODA loop (observe, orient, decide, act) training to the developers. The OODA loop process is intended to increase situational awareness. Developers again recorded their errors over the five days following the training. Nagaria & Hall identified seven themes among reported developer errors (*e.g.* poor internal/external communication, poor code structure/increased code complexity, over-complicated development environments, errors in sequence-based tasks, syntax errors, and special cases). Developer responses and self-reported errors indicate that OODA loop training (1) is perceived as useful and (2) can reduce the number of errors made by software developers on a daily basis. Then, Nagaria & Hall mapped reported errors to slips (74 total), lapses (71 total), and mistakes (30 total), but they do not discuss these classifications in detail, except to note that communication-based errors tend to be mistakes, rather than slips or lapses. Nagaria also summarized this study in their doctoral dissertation [251].

Other work using Reason's GEMS framework to study human error in computing includes research by Hu *et al.* which suggests that using HET to find software faults makes developers less likely to introduce those faults during future requirements engineering [140, 141], and a study by Manjunath *et al.* in which professional software engineers suggested improvements to HET/HEAA training and recommended mitigations (*e.g.* creating a communication plan, consulting subject-matter experts) to prevent human errors in requirements engineering [194]. Note that the work by Hu *et al.* and Manjunath *et al.* was in collaboration with Vaibhav Anu.

3.1.3 Errors of Omission and Commission

In 1986, Basili & Rombach [30] classified software faults as omissions (*i.e.* the fault stems from something missing) or commissions (*i.e.* the fault stems from something that is incorrect) as part of a methodology for improving software engineering processes by tailoring those processes to the specific goals of the project. Examination of a single project using the proposed methodology revealed that the majority of faults (76%) were commission faults, which aligns

Table 3.6: Omission and Commission Requirements Defects from Kirner & Abib [165]

Human Error	**Category**
Missing Functionality	Omission
Missing Environment	Omission
Missing Performance	Omission
Missing Interface	Omission
Ambiguous Information	Commission
Inconsistent Information	Commission
Incorrect Fact	Commission
Wrong Section	Commission

with observations made by Marick [200] and Basili & Perricone's previous findings [29]. Marick [200] further notes that while the majority of faults are errors of commission, omission errors are still important. Basili notes that omission faults are harder to detect with source code-based detection methods/tools (*e.g.* structural testing) than with functional testing or code review.

Glass [115] studied faults in two military aircraft software systems (in total, 600,000 instructions written by 180 developers). Based on post-delivery problem reports, Glass found that omitted logic faults (*i.e.* code with missing functionality) were the most common fault to make it into production (as opposed to inadequate requirements, referencing the wrong variable, timing, and other faults). Similarly, Ostrand & Weyuker [273] reported that 55% of faults stemmed from omitted code and, of the faults stemming from a decision, 81% were the result of omitted code. These findings are contrary to previous findings by Basili *et al.* [29, 30].

Wright *et al.* [364] described a process to analyze interactions between users and a software system as a method to derive human error based requirements for the behavior of software systems. While errors of omission and commission are described and discussed, Wright *et al.* suggests that their process is human error theory agnostic.

Kirner & Abib [165] inspected software requirements specifications and found that the majority of defects (84% in an ATM system and 92% in a parking garage control system) were commission defects. Kirner & Abib described four types of omission defects and four types of commission defects for requirements documents. Omission defects in requirements documents included missing functionality (*i.e.* missing information about the desired internal operation behavior of the system), missing environment (*e.g.* missing details about required hardware, software, and/or database systems), missing performance (*i.e.* missing information about the desired performance specification), and missing interface (*i.e.* missing details about how the system will interact with external systems). The types of commission defects outlined by Kirner & Abib were ambiguous information, inconsistent information, incorrect facts (*i.e.* assertion of a fact that cannot be true based on the software requirements specification), and wrong section (*i.e.* misplaced information).

Bass *et al.* [32] evaluated 18 software architectures with the goal of discovering patterns in risk themes (*i.e.* common types of risks related to software architecture, process, and organization) identified by those evaluations. Contrary to findings for commission/omission faults/defects in earlier work, Bass *et al.* found that the majority of 99 risk themes were classified as omissions (*i.e.* risk themes resulting from a failure to perform necessary activities), rather than commissions (*i.e.* risk themes resulting from problematic decisions in the architecture). Bass *et al.* only summarize the classification of omission/commission risk themes without providing examples.

Park *et al.* [279] examined open source Java bugs that were fixed multiple times to understand why omission errors occur and how they can be prevented. The noteworthy contribution of this work is that previous works [106, 164, 256, 374] focused on detecting omission errors, while Park *et al.* sought to understand what kinds of omission errors occur. To that end, Park *et al.* presented a list (which they refer to as a taxonomy) of omission errors for bugs that required supplementary fixes. The omission errors that Park identified include: porting initial patches to another OS/system/branch, incorrect conditional statements, failing to update subclasses of the same type at the same time, incomplete refactoring, missing null-pointer checks, and others.

Itkonen *et al.* [148] studied the role of a tester's knowledge in exploratory software testing. Software failure symptoms were classified as omissions or commissions. Commission failures were further classified by how their symptoms manifested; failures were observed either in the presentation/layout (*e.g.* on screen or in report printouts), as error messages, as extraneous functionality, as inconsistent states, or as incorrect results. Omission failures were also further classified based on how they were observed, either in the presentation/layout (*e.g.* missing visual elements), as missing functionality, as lack of feedback (*i.e.* expected feedback from the system is not given), or as lack of capability (*i.e.* some part of a function/feature is missing). As with most findings in earlier work, the majority of failures were commissions.

Niu *et al.* [260] proposed a scenario-based method "to assess how software architecture affects the fulfillment of business requirements [260]" to help developers choose an appropriate software architecture for their needs. Niu *et al.* classified risks when completing non-functional requirements as commissions (*i.e.* caused by suspicious decisions related to the system architecture), omissions (*i.e.* caused by unfinished activities), and others (*i.e.* not omission or commission errors). This work found that risks of commission and omission occur in equal numbers, noting that this is an unexpected result based on the findings of Bass *et al.* [32].

The work of Santos *et al.* [315, 316, 317] catalogs architectural software weaknesses using the concepts of omission (*i.e.* missing a necessary security tactic) and commission (*i.e.* choosing an incorrect security tactic) in addition to *realization* (*i.e.* incorrectly implementing a properly chosen security tactic). We have not encountered the concept of realization in other works using omission and commission. Specific classifications for omissions and commissions are not available.

In 2018, Lin *et al.* [187] studied the programmatic syntax of bugs in the Defects4J bug repository [157] and found that 46.4% of bugs were omissions (which they define as faults caused by missing the execution of some code). Lin *et al.* divided omission bugs into data omissions (*e.g.* missing variable assignments, incorrectly evaluated conditionals, incorrect boolean logic in conditionals statements, missing function calls) and control omissions (*e.g.* missing return statements, missing `if` blocks, missing exception `throw` statements, missing exception handling, calling an incorrect function, passing the wrong parameter to a function). The impact of specific omission categories on software engineering is not discussed.

Other research has considered errors of omission and commission for safety analysis for complex programmable electronic systems [276], evaluation of faults in system architectures [353], general software safety/reliability engineering [277, 355, 356], and software evaluation [319].

3.2 Human Error Assessment

An estimated 60-90% of system failures result directly from human error [132]. Indeed, human error is a primary cause of failure in the nuclear [298], chemical [158], aviation [321], and maritime [304] sectors. As a result, human error assessment (HEA) has been an ongoing area of interest. Human reliability analysis (HRA) is one form of HEA that aims to determine the impact of human error on a system [335] by systematically assessing the probability of human error based on human actions and decisions [124]. HRA typically involves five stages: (1) defining the problem and scope of analysis, (2) task modeling, (3) analysis of human errors, (4) human error quantification, and (5) recommendations for error management [134]. HEA has taken other forms over the years, such as probabilistic safety assessment for nuclear power plants [4] and human reliability analysis in manufacturing [79] and medicine [193].

A related concept is After-Event Reviews (AER)—processes for learning from experience. AERs enable learners to "systematically analyze the various actions that they selected to perform a particular task, to determine which of them was wrong or not necessary, which should be corrected, and which should be reinforced [86]." AERs typically involve a facilitator (*e.g.* instructor) who aids the learner in trying to understand why their actions led to a specific outcome. AERs enable learners (and groups of learners) to (1) reflect on their performance, (2) understand why expected outcomes/objectives were not met, (3) identify lessons learned from past experience, and (4) evaluate how lessons learned can be incorporated and internalized to improve performance [86]. Put simply, AERs are a form of guided self-reflection, which helps learners become more aware of their personal experiences and thus learn from them [10, 120]. AERs have been shown to change individuals' mental models in four ways: "by intensifying self-explanation, by advancing data verification, by providing process feedback, and by enhancing self-efficacy [86]." AERs have been applied to enable learning in the military [6], for learning from wildfires [75], for learning in the workplace [107], and to learn from errors during motorcycle production [118].

Software quality improvement has traditionally been focused on software faults [18]. Root cause analysis (RCA)—tracing faults to their origin—was developed to identify the first software fault in a chain of faults [183, 202]. The goal of RCA is to enable software engineers to modify their processes to eliminate the first fault, thus preventing the whole fault chain [18]. However, RCA is time-consuming, so researchers developed orthogonal defect classification (ODC), a structured process of identifying the trigger for a fault, instead of tracing the full chain of faults [62]. ODC is less time-consuming than RCA, but relies on statistical analysis and large datasets of fault reports, which may not be available or feasible [18]. The notable shortcoming of RCA and ODC is that neither approach can identify the human error behind the fault [18]. In 1998, Lanubile *et al.* [181] shifted from emphasizing faults to focusing on errors, faults, or flaws "in the human thought process that occurs while trying to understand given information, while solving problems, or while using methods and tools [18]." This shift motivated Anu *et al.* [12, 15, 18, 22, 23].

As discussed in Section 3.1.2, Anu *et al.* created the HET and HEAA [12, 15, 18, 22, 23] to assess human errors during the software requirements phase, and evaluated their approach with a series of studies. Results were favorable, showing improved fault detection using HET [16, 19]. The work by Anu *et al.* is, however, the only research we found that examines human error assessment in software engineering. But software engineers are not unfamiliar with post-mortem reflection.

In addition to traditional fault inspection (*e.g.* RCA, ODC), there are a variety of post-mortem reflection processes. For example, Microsoft at one point employed post-mortem reports, discussions of what worked well in a project, what didn't work well, and what the team could do to improve the next project [69]. Tiedeman [339] discusses three types of post-mortems in software engineering: (1) planning post-mortems to assess requirements elicitation, (2) design/verification post-mortems following software design and implementation, and (3) field post-mortems to assess the project some time after deploying it to production.

Dingsøyr [80] summarizes three proposed methods for conducting post-mortems in software engineering from Whitten [359], Collison & Parcell [66], and Birk *et al.* [40, 81, 85, 162, 331]. These three approaches have five shared aspects, which we considered when designing our human error informed micro post-mortem process in Section 6.1:

- Select relevant (ideally objective) participants; exclude managers
- Identify what went well and why
- Identify what didn't go so well, including any challenges faced by project team members
- Identify (and implement) improvements for future projects
- Document the post-mortem

Dingsøyr [80] goes on to suggest some requirements for a good post-mortem process:

- **For participants:** openness, patience, politeness, the ability to listen, and courage
- **For facilitators:** a skilled leader who encourages open dialogue and establishes a good atmosphere
- **For the process:** adequate time—this varies based on company goals, project size, number of participants, and other factors—and an atmosphere of safety

3.3 Software Engineering & Security Taxonomies

Various security-centered taxonomies have been proposed since at least the 1970s [192]. These taxonomies primarily catalog two aspects of security—vulnerabilities (and weaknesses) and attack patterns—and serve as tools for vulnerability discovery and assessment, guidance for engineering more secure software, and formal security education. Another goal of security taxonomies is to provide a common language for the study of vulnerabilities and attacks [135, 235]; however, there is no universally accepted gold standard [146]. The subsections that follow provide a summary of proposed vulnerability and attack taxonomies, informed heavily by the work of Lough [192] and Igure & Williams [146]. We make no claim that the taxonomies discussed below form an exhaustive list. Note that Section 3.3.1 also discusses a handful of noteworthy vulnerability databases/lists, which are not taxonomies.

3.3.1 Vulnerability Taxonomies

In 1974, McPhee [205] developed one of the first vulnerability lists (not a true taxonomy [146]), but it was very limited in scope, collecting only seven integrity flaws (*e.g.* concurrent use of serial resources, user data passed as system data) in IBM's OS/VS2. Another early attempt was that of Attanasio *et al.* [26], who studied the results of penetration testing experiments and collected a list of 16 OS features (based on 35+ flaws) likely to have flaws (*e.g.* add-on features, error handling, violating design principles).

Perhaps the first true vulnerability taxonomy was produced in 1976 by the Research in Secured Operating Systems (RISOS) project, which placed 18+ vulnerabilities into seven classes of vulnerabilities in operating systems (*e.g.* incomplete parameter validation, authentication/authorization) [1]. The Protection Analysis (PA) project was very similar to RISOS, but yielded 10 vulnerability categories (*e.g.* interrupted atomic operations, data consistency over time) from 100 vulnerabilities in various OSs (*e.g.* TENEX, MULTICS, UNIX) [41]. Both of these taxonomies organize vulnerabilities into categories based on operations or functions in the OS.

The Stanford Research Institute analyzed over 350 security breaches and placed them into one of seven major categories: intentional violation—internal; intentional violation—computer department; intentional violation—external; aura of computer; disaster; accidents; and miscellaneous. These seven categories are further divided into 71 subcategories (*e.g.* insertion of data, forced entry, disclosure of data) based on the type of violation that occurred [257]. The noteworthy contribution of this work is that seven categories of violations are also related to four major categories of safeguards—management, system, industrial security, and educational/legal—in a matrix. These four categories are further divided into 34 sub-categories (*e.g.* audit practices, hardware monitors, password controls, storage and backup).

Moving into the 1980s, we could not find any work in the area of vulnerability taxonomies, but there was work with general bug taxonomies, including Ostrand & Weyuker's, which placed 173 faults from UNIVAC into seven categories [273], and Knuth's, which classified 850 bugs from TEX into 15 categories [167]. Knuth's taxonomy is noteworthy, as it is comprehensive and unambiguous.

In 1990, a textbook by Beizer [37] proposed a taxonomy of software bugs categorized by when they are introduced into software, either during design, implementation, or maintenance. Landwehr *et al.* [180], in 1994, proposed three distinct vulnerability taxonomies based on how the vulnerability was introduced, when it was introduced (in terms of the software development lifecycle), and the location of the program in the system. Jiwnani & Zelkowitz [152, 153] combined the three taxonomies from Landwehr *et al.* to build a matrix relating 1,013 vulnerabilities based on their causes, locations in the program, and their impact. Du & Mathur [84] also adapted Landwehr *et al.*'s taxonomy, classifying 150 vulnerabilities into three categories (and 15 subcategories), based on their cause, impact, and fix.

In 1995, Bishop [42, 43] reworked the RISOS and PA taxonomies, classifying vulnerabilities based on six characteristics: nature of the flaw, introduction time, exploitation domain, effect domain, minimum number of components required for exploit, and identification source. In the same year, Aslam [25] proposed a taxonomy of security flaws in UNIX systems, with three high-level fault categories—operation, environmental, and coding faults—and 14 specific faults, including syntax errors and race conditions. Krsul [174] extends Aslam's taxonomy, focusing mainly on environmental assumptions and classifying 210 vulnerabilities. Richardson [306, 307] adapts and expands on Aslam's and Krsul's work to create a vulnerability database for Denial-of-Service (DoS) attacks, containing 630 attacks spanning categories such as brute force and data poisoning.

In 1997, Jayaram & Morse [149] organized security threats to networks into five categories: physical, system weak spots (exploited for unauthorized access), malign problems (*i.e.* placing malicious code in the system), access rights (*i.e.* spoofing legitimate users), and communication-based (*e.g.* eavesdropping).

While not a true taxonomy, perhaps the most widely used vulnerability database today is Common Vulnerabilities and Exposures (CVE), first conceived in 1999 [195]. At the time of writing, CVE has collected over 208,000 vulnerabilities spanning 25 years (1999-2023) [237]. Each CVE entry contains a description of and references (*e.g.* security advisories, bug reports) to the vulnerability. For example, CVE-2020-9371 describes a Cross-Site Scripting (XSS) vulnerability in a WordPress plugin [234]. While CVE's goal is to "identify, define, and catalog publicly disclosed [237]" vulnerabilities, it is not a taxonomy because it lacks any classification of vulnerabilities. The National Vulnerability Database (NVD) [46], created in 2000, expands on CVE by including Common Vulnerability Scoring System (CVSS) [207] severity scores and a mapping to Common Weakness Enumerations (CWE) [235] entries.

Knight [166] proposed a vulnerability taxonomy in which each vulnerability was defined by its fault, severity, consequences, the level of authentication necessary to exploit the vulnerability, and the tactic used to exploit the vulnerability. These vulnerability properties are still used in modern vulnerability assessment in tools such as the CVSS [207] and the Common Weakness Scoring System (CWSS) [201].

Since 2003 [360], the Open Web Application Security Project (OWASP) has maintained a ranking of the top ten security risks for web-based applications [274], but this is not a true taxonomy.

In 2003, Wang & Wang [354] released a taxonomy focusing on security risks in the application-layer (*e.g.* credential theft, data exposure), platform-layer (*e.g.* unauthorized administrative access), and network-layer (*e.g.* DoS, network traffic exposure). In the same year, Gray [119] extended the work of Landwehr *et al.* [180], Bishop [42], and Wang & Wang [354] to develop a new vulnerability taxonomy, including details about the method of discovery (*e.g.* manual or automatic source code analysis), exploit detection, and mitigation strategies.

In the following year, Pothamsetty & Akyol [286] presented a taxonomy for vulnerabilities in network protocols with seven categories, including clear text communication (*i.e.* lack of strong encryption), non-robust message parsing (protocols that handle unexpected properties of messages poorly), lack of authentication mechanisms, and entropy problems (*i.e.* insufficiently strong pseudo-random number generating algorithms). A strong contribution of this work is the inclusion of testing techniques (*e.g.* protocol field fuzzing, packet sniffing) and metrics (*e.g.* CPU utilization, amount of logging to the console), as well as some best practices for engineering secure protocol software (*e.g.* validating buffer/input data, rate limiting).

In 2005, Tsipenyuk *et al.* [347] presented Seven Pernicious Kingdoms, a taxonomy of security errors, which contained seven categories related to source code errors and one category for environmental/configuration errors. Notably, Tsipenyuk *et al.* listed the categories in "order of importance to software security [347]:" input validation and representation, API abuse, security features, time and state (*i.e.* concurrency errors), errors (related to exception handling), code quality, encapsulation, and environment. The motivation behind seven (plus one) categories is that humans have a limited working memory; on average, humans can remember seven (plus-or-minus two) things at a time [223]—restricting the taxonomy to eight categories keeps it simple for security practitioners to use.

CWE, first released in 2006 [236], collects the (mostly software-agnostic) underlying weaknesses that lead to a CVE. The goal of CWE is to be "a baseline for weakness identification, mitigation, and prevention efforts [235]", as well as a resource for security tools. Each CWE entry includes a wealth of information, including short and extended descriptions, background details, consequences of vulnerabilities based on the weakness, and mitigation strategies. CWE is organized in a hierarchy of related weaknesses; top-level parents (called pillars) have related child classes (smaller categories) and bases (specific weaknesses). For example, the CWE entry describing the underlying weakness (XSS) of CVE-2020-9371 would be CWE-79 [240], which is a type of injection weakness (CWE-74 [239]), falling under the top-level pillar CWE-707: Improper Neutralization [238].

In 2010, Howard *et al.* [137] suggested 24 *deadly sins* of software security, a collection of programming flaws with security consequences, grouped into four categories: web application (*e.g.* SQL injection, hidden form fields), implementation (*e.g.* command injection, information leakage, poor usability), cryptographic (*e.g.* incorrect usage of or inadequate cryptography), and networking (*e.g.* unencrypted network traffic, improper use of public key infrastructures) sins. While not presented as a taxonomy, this work contains a wealth of taxonomic information, such as links to CWE, information on affected languages, steps for identifying and testing flaws, and defensive measures.

Hajian *et al.* [126] presented a taxonomy of network vulnerabilities organized by their location/activities (*e.g.* network infrastructure, network services, network-based applications, management activities, control activities, end-user activities), cause (*i.e.* CWE entries), and impact (*e.g.* confidentiality, availability, privacy, authentication).

Zhao & Dai [375] proposed categorizing vulnerabilities into a taxonomy based on five information security attributes: confidentiality, integrity, availability, controllability, and reliability. While interesting, the taxonomy suffers from subjective ranking (high-medium-low scale) within categories.

In 2019, Alkhalifah *et al.* [5] studied 65 security incidents involving blockchain to create a taxonomy of threats and vulnerabilities specific to blockchain, however the details of this taxonomy are not yet available.

The Vulnerability History Project (VHP) [209] is a museum of security mistakes curated by Dr. Andrew Meneely and computing students at the Rochester Institute of Technology (RIT). VHP collects CVE entries for large open-source software projects, including Chromium, Django, ffmpeg, and some Apache products. Software engineering students and student researchers at RIT dive deep into each CVE to improve descriptions and CWE mappings, outline specific mistakes that were made by developers, and fill in missing information, such as applicable bug bounties, discovery dates, how vulnerabilities were discovered, and the commits where vulnerabilities were introduced and fixed. While not presented as a taxonomy, the VHP website does categorize vulnerabilities based on programming language, the size of the vulnerability fix, severity metrics, vulnerability lifetime, CWE, bounty information, and other characteristics.

3.3.2 Attack Taxonomies

Perhaps the first attack taxonomy was that of Perry & Wallich [283] in 1984, which organized computer attacks as a two-dimensional matrix. On one axis was one of six types of users: operators, programmers, data entry, internal users, outside users, and intruders. The second axis had six types of computer crimes: physical destruction, information destruction, data diddling (*i.e.* altering data), theft of services, browsing, and theft of information. Unfortunately, these types of users, representing sources of attack, are not mutually exclusive [146].

Neumann & Parker developed the SRI Computer Abuse Methods Model based on the analysis of almost 3,000 computer misuse cases spanning almost 20 years [254, 255, 280, 281]. This resulted in nine categories of misuse that encompass 26 types of attacks, such as shoulder surfing, physically damaging hardware, spoofing people and accounts, virus- and malware-based attacks, and side-channel attacks [254]. Lindqvist & Jonsson [188] expanded on three categories from Neumann & Parker [254], adding seven attacks (*e.g.* automated searching, resource exhaustion).

In 1997, John Howard categorized six years of Community Emergency Response Team (CERT) incidents into a taxonomy based on the types of malicious actors, the tools they use, information on access levels and accessed information, and the goals and results of an attack [136]. The following year, Howard & Longstaff [135] presented a similar taxonomy, focused on the tools used in the attack, the vulnerability exploited in the attack, the actions taken by the attacker, the intended target of the attack, and the outcome of the attack. However, Howard's taxonomy suffered shortcomings, notably that its categories were not mutually exclusive. The Sandia Laboratory Taxonomy [63] was very similar to Howard's, but suffered from the same shortcomings.

In 1997, Ranum [292] presented a taxonomy of attacks with eight categories based solely on the techniques used by the attacker: social engineering, impersonation (*e.g.* network sniffing to obtain valid credentials), exploits (*i.e.* attacks that exploit a specific vulnerability), transitive trust (*i.e.* exploiting host-to-host or network-to-network trust), data driven (*e.g.* viruses, malware), infrastructure (*e.g.* DNS spoofing, ICMP bombing), denial of service, and magic (*i.e.* attacks not invented/observed yet).

Weber [357] proposed a taxonomy of computer intrusions based on the level of privilege required for the attack to succeed (*e.g.* root access, remote network access), the means by which the attack is executed (*e.g.* abusing features, exploiting bugs), and the intended impact of the attack (*e.g.* DoS). Weber's taxonomy was later adapted by Lippmann *et al.* [189, 190], including categories based only on the impact of the attack.

In his Ph.D. dissertation, Lough sought to reconcile the limitations of these previous attack taxonomies (and others not mentioned here) by proposing his own VERDICT: Validation Exposure Randomness Deallocation Improper Conditions Taxonomy, based on commonalities of previous taxonomies [192], but VERDICT was not widely adopted.

Welch & Lathrop [358] presented the Wireless Threat Taxonomy used to build security into the wireless network at West Point, which organized security threats into seven categories: traffic analysis, passive and active eavesdropping, unauthorized access, man-in-the-middle, session hijacking, and replay attacks.

In 2004, Killourhy *et al.* [163] proposed a defense-centered attack taxonomy, which organizes attacks based on the way they manifest as anomalies in sensor data (*e.g.* how they appear to IDS systems). Through examining sensor data and known attacks, Killourhy *et al.* observed four categories of attack manifestations: system calls that never appear in normal records, minimal foreign sequences (system calls that never appear normally, but subsequent calls that do), sequences that only partially match normal ones, and sequences that completely match normal ones.

Hansman & Hunt [127] presented a taxonomy of network and computer attacks with four dimensions: attack vector (*e.g.* viruses, buffer overflows, DoS), target of the attack (from general OS to specific software versions), the vulnerabilities and exploits used in the attack, and the capability of the attack to have an impact beyond itself (*e.g.* a virus that causes harm, but also installs a Trojan horse; the Trojan horse has an impact beyond the initial virus). Hansman & Hunt also discuss other potential dimensions, including relative damage from the attack, cost to clean up after the attack, and defensive strategies for an attack.

Yu *et al.* [369] placed types of attacks into 10 categories, including authentication management (*e.g.* default passwords, brute force), information disclosure, input manipulation (*e.g.* XSS, SQL injection), and cryptographic (*e.g.* weak or missing encryption). Yu *et al.* also mapped vulnerabilities to each category of attack, making this a sort of hybrid taxonomy [146].

Introduced in 2007, Common Attack Pattern Enumeration and Classification (CAPEC) [233] is a catalog of over 500 common attacks intended to help security practitioners understand how malicious actors exploit weaknesses in software. CAPEC entries include information on prerequisites for the attack, consequences of the attack, and mitigations for the attack, as well as mappings to related CWE entries.

Lai *et al.* [177] proposed a taxonomy of web attacks with dimensions focused on the type of attack (*e.g.* zero-day, session hijacking), the HTTP method involved (*e.g.* `GET`, `POST`, `DELETE`), the web server software (*e.g.* Apache, IIS), the markup language used (*e.g.* ASP, PHP), and the resulting damage (*e.g.* remote access, resource exhaustion).

In 2009, Meyers *et al.* [221][2] proposed an attack taxonomy with nine non-mutually exclusive categories of attack based on the type of attack (viruses, worms, Trojans, buffer overflows), the goal of the attack (DoS, information gathering, password/account compromise), and the attack vector (network, physical).

In 2013, MITRE released the first version of ATT&CK™ [334], which currently documents over 200 techniques (*e.g.* phishing, network service scanning, and data obfuscation) that malicious actors use during an attack. Techniques are collected in 14 categories (called tactics), including reconnaissance, discovery, and lateral movement. Tactics and techniques are intended to describe why and how the attacker performed an action, respectively. In 2021, MITRE released the complementary D3FEND™ [241], a taxonomy of countermeasures and mitigations. If we consider security a game between offense (malicious actors) and defense (security practitioners), ATT&CK™ catalogs actions taken by the offense, while D3FEND™ describes actions taken by the defense.

Simmons *et al.* [326] proposed an attack taxonomy called AVOIDIT (Attack Vector, Operational Impact, Defense, Information Impact, and Target), with five major categories: attack vector (*e.g.* kernel flaws, race conditions), operational impact (*e.g.* DoS, installed malware), defense (both in terms of mitigation and remediation), information impact (*e.g.* disruption, disclosure), and target of the attack (*e.g.* network, user, database).

Juliadotter & Choo [154] proposed an attack and risk assessment taxonomy for cloud-based services, classifying attacks based on aspects of the attacker (*e.g.* skill level, motivation), the attack vector (*e.g.* ease of exploit and discovery), the target of the attack (*e.g.* data centers, networks, accounts), the impact of the attack, and defensive measures against the attack.

[2]No relation to the author of this work.

While not purely an attack taxonomy, Rizvi *et al.* [308] presented a security taxonomy for Internet-of-Things (IoT) systems that included details on attacks. This taxonomy outlined the architectural layer where a vulnerability may exist (*e.g.* data collection, application, and network layers), the types of attacks that could occur (*e.g.* communication-based attacks such as DoS, physical attacks like reverse engineering and radio interference, and software-based attacks such as exploiting misconfigurations), the trust that the user has in the IoT system (*e.g.* privacy of user data, the availability of the system, and the reliability of transmitted information), and compliance of IoT systems with policies and organizational oversight (both government and non-government oversight). While significant work went into developing this taxonomy intended to help researchers identify security challenges in IoT and existing solutions to those challenges, Rizvi *et al.* did not demonstrate its application to real-world IoT security systems.

3.3.3 Miscellaneous Security Taxonomies

Aside from categorizing vulnerabilities or attacks, another aspect of security that has given rise to taxonomies is the attackers themselves. Landreth & Rheingold [179] made one of the first attempts to classify different types of hackers as novices (*i.e.* entry-level mischief-makers), students (*i.e.* academics hacking for the challenge with minimal malicious intent), tourists (*e.g.* thrill-seekers), crashers (*i.e.* those hacking for pride), and thieves (*e.g.* true criminals hacking for profit). Later, Hollinger [131] grouped hackers into pirates (*i.e.* minimally technical hackers focused on pirating copyrighted material), browsers (*i.e.* casual hackers who enjoy browsing people's private files), and crackers (*i.e.* serial offenders focused on sabotage). Chantler [61] suggested three types of hackers: losers/lamers (*i.e.* minimally technical attackers motivated by greed/vengeance), elites (*i.e.* highly skilled hackers motivated by excitement/achievement), and neophytes (*i.e.* semi-skilled learners trying to achieve elite status). While all interesting categorizations, these were not true taxonomies.

Perhaps the first true taxonomy of attackers was presented by Rogers [313, 314], who studied 66 cyber-adversaries, documenting social/demographic characteristics such as gender, race/ethnicity, relationship status, and education level. Following Rogers' work, Meyers *et al.* [221][3] placed attackers into eight adversary classes and described the skill-level, level of malicious intent, motivations, and methods of attack of attackers in each class. The eight classes include novices (*e.g.* script kiddies, newbies), hacktivists (*i.e.* political activists), crashers/cyber punks (*i.e.* those motivated by thrill), insider threats, coders (*i.e.* script/tool writers and mentors), white hat hackers (*i.e.* non-malicious hackers), black hat hackers (*i.e.* professional, malicious hackers), and cyber-terrorists.

Other security related taxonomies that have been proposed include a taxonomy of information security risk assessment approaches [323], a taxonomy of operational security risks (*e.g.* the actions of people, external events, and technology failures) [58], a taxonomy of IDS systems [27], a taxonomy of exploit monitoring approaches [322], a hybrid taxonomy of attacks and vulnerabilities in embedded systems [278], a taxonomy of botnet detection techniques [371], a taxonomy of network fuzz testing techniques [248], a taxonomy of collaborative security mechanisms [210], a taxonomy of bug tracking process smells [290], and a requirements taxonomy based on website privacy policies [11].

3.4 Apology Mining

Apologies in natural language have been studied in socio-linguistic contexts [47, 65], including dialog act research [342], and used as features in machine learning contexts, such as assigning politeness scores to natural language [70] and mining software engineering artifacts for emotions [249]. However, we found little work using apologies directly to study software developers and/or software development. This suggests a new area of study for software engineering research.

Li *et al.* [184] examined developers' negative sentiment in open source software projects on GitHub and outlined nine negative sentiment-related events, including apologies, which they define as "apologizing or expressing guilt due to delays in work progress or inconvenience inflicted on other developers [184]." Similar to our work, keywords (*e.g.* *apologize*, *sorry*, *apology*, *pardon*) were used to identify apologies for a rule-based classifier (0.906 accuracy), but Li *et al.* did not use lemmatization or provide a full list of keywords.

[3]No relation to the author of this work.

Chapter 4

Systematization of Software Engineers' Human Errors

In this chapter, we describe two studies that systematically identify human errors experienced by software engineers. The first study (Section 4.1) outlines a systematic literature review of human error research in the software engineering domain, which leads to Version 1 of T.H.E.S.E. The second study (Section 4.2) outlines a process for identifying human errors from software engineers' natural language on GitHub, leading to Version 2 of T.H.E.S.E.

4.1 Systematic Literature Review of Human Errors in Software Engineering

4.1.1 Motivation & Research Questions

Software engineering is a complex process of gathering requirements from stakeholders, designing a software system, and implementing, testing, deploying, and maintaining that software. Each phase of the development lifecycle encompasses many activities, including risk assessment, threat modeling, code review, a wide range of testing, responding to bug reports, patching vulnerabilities, and deploying updates—all while communicating with peer developers, managers, stakeholders, and users. No complex process is inherently safe [76], so software engineers, despite their best efforts, inevitably make mistakes.

Mistakes, or **human errors**, have been extensively documented and studied in psychology research for well over 50 years. Sigmund Freud drew attention to human error in 1901 as he studied slips of the tongue, forgetfulness, and omissions [105]. In 1937, Kollarits conducted one of the first studies of human error, examining about 1,200 human errors experienced by himself, his wife, and colleagues. Kollarits proposed at four categories of human error: substitution, omission, repetition, and insertion [169]. Later, in the 1980s, Jens Rasmussen studied human error primarily in the context of industrial accidents [294, 296], and classified human errors as skill-, rule-, or knowledge-based [293, 295]. Building on Rasmussen's work, James Reason categorized human errors under his Generic Error-Modelling System (GEMS) [298] as slips, lapses, and mistakes (failures of attention, memory, and planning, respectively).

Software engineers have also been concerned with human errors (albeit indirectly), defining, studying, and cataloging the consequences of their human errors: defects (software faults and failures) [48, 145], weaknesses, bugs, and vulnerabilities. With open source software, software engineers have documented their mistakes and put them on display, adopting vulnerability disclosure as an industry standard so that current and future developers can learn from their mistakes. The Common Vulnerabilities and Exposures (CVE) database [237], for example, provides a timestamp and a brief description of thousands of documented vulnerabilities, along with links to related security advisories, changelogs, bug reports, patches, and severity measures. Further, the Common Weakness Enumeration (CWE) [235] taxonomizes vulnerabilities into technical faults and their mitigations. While the CVE and CWE are not perfect, they are systematic endeavors which enable learning from critical software security mistakes. Vulnerability assessment is a noble and valuable endeavor, but it's only one piece of the puzzle; software faults are caused by human errors [349].

However, adoption of human error research from psychology into software development has been relatively slow and short-lived (except for work by Anu *et al.* [12, 15, 18, 22, 23]). While some software developers and researchers have paid attention to human error, Wood & Banks expressed concern in 1993 that "many contemporary information security practitioners appear to have forgotten about [362]" human error. 12 years later, Im & Baskerville expressed "the serious need for research and practical knowledge about the management of human error in secure information systems [147]." So if human errors are major concerns, where is human error assessment in software engineering? To date, post-mortem assessment of software engineers' defects is widely adopted in the software security domain. Figure 1.1 shows visually how human error has a lasting impact on software development, leading to faults in code, which manifest as software failures. Failures are reported, and then their faults patched, but the current landscape of software engineering does not take their faults and failures a step further to confront their underlying human errors. Our broad aim is to build upon the practice of vulnerability assessment by making human error assessment as well-defined and accessible to software engineers.

With that goal in mind, we address the following research questions regarding discovery, appraisal, and usefulness (as suggested by Brereton *et al.* [50]):

RQ 1: **Human Error Discovery**
What human errors experienced by software engineers are documented in previous research?

RQ 2: **Human Error Appraisal**
What are the strengths and weaknesses of previous research about human error in software engineering? We define strengths and weaknesses in terms of software engineering scope, category ambiguity, and category completeness.

RQ 3: **Human Error Usefulness**
How comprehensive is existing research about human errors in software engineering? We define usefulness in terms of coverage of human error theories from cognitive psychology and coverage of human errors in software engineering activities.

4.1.2 Methodology

This section outlines our methodology for identifying literature related to human errors in software engineering, selecting relevant literature, and reviewing selected literature. Brereton *et al.* [50] analyzed systematic literature reviews (SLR) in software engineering and outlined the key components of a SLR, which we adopted into our methodology.

We began by specifying our research questions, which Brereton *et al.* identifies as the most critical step in any SLR [50]. Our study was exploratory, with the goal of identifying human errors documented in previous software engineering research (*i.e.* **RQ 1**). Section 4.1.2.1 describes our review protocol, *i.e.* our process for identifying primary studies [50]. In Section 4.1.2.2 we outline the information we extracted and how we assessed the quality of selected literature (*i.e.* **RQ 2**). Finally, in Section 4.1.2.3 we describe our process for synthesizing the human errors from existing software engineering literature into a Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.) (*i.e.* **RQ 3**). Our SLR was originally conducted from June 1, 2021 through November 30, 2021, and re-conducted from October 2, 2022 through January 7, 2023 to identify newly published work on human error in software engineering.

4.1.2.1 Systematic Literature Review Protocol

For our SLR, we searched for literature in Google Scholar[1], IEEE Xplore[2], and the ACM Digital Library[3]. For IEEE Xplore and the ACM Digital Library, we formed search queries using boolean logic based on five domain strings and 19 scope strings, as outlined in Figure 4.2. Figure 4.3 shows the exact boolean logic query we used. We attempted to use the same boolean logic query for Google Scholar, but the search results were (more often than not) unrelated to software engineering or human error (*e.g.* the first page of search results contained papers related to bankruptcy, computing accreditation, corporate taxes, civil service, and law). As a result, we used a more targeted boolean logic query for Google Scholar, as shown in Figure 4.4. Figure 4.1 contains a visual summary of our SLR process.

We did not restrict our search to any specific time period or any specific type of literature (*e.g.* short vs. long papers, journal vs. conference vs. book). We did not change any of the default search options for any search engine. For each search engine, we examined search hits to determine whether they were relevant to human errors in software engineering with a three-stage process:

[1]`https://scholar.google.com/`
[2]`https://ieeexplore.ieee.org/Xplore/home.jsp`
[3]`https://dl.acm.org/browse/`

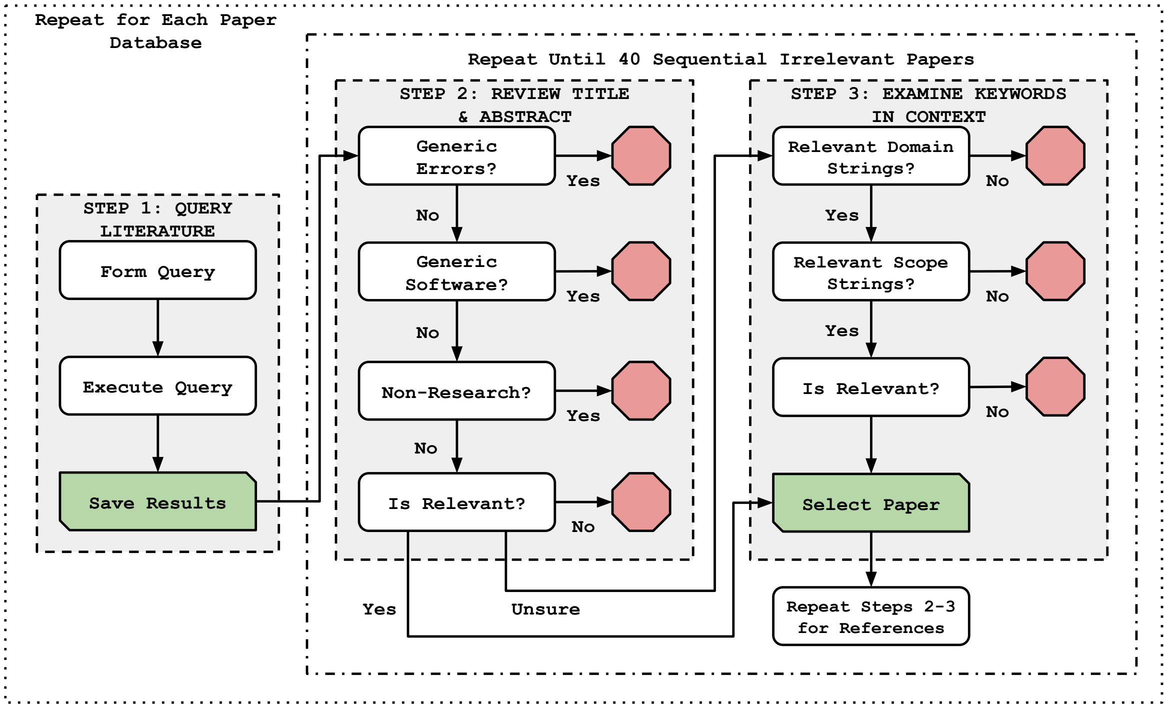


Figure 4.1: Visual Summary of Our Systematic Literature Review Process

Step 1 of our SLR involved querying the literature. In Step 2 we reviewed the title and abstract of each search result to determine relevancy to human error in software engineering. When the title and abstract did not provide enough information to determine relevancy, we examined keywords within the body of the paper in Step 3. For selected papers, we repeated Steps 2 and 3 for references to find more relevant work. Steps 2 and 3 were repeated for each of the three search engines we queried.

1. **Review Title & Abstract:** We read the title and abstract of each paper and determined whether or not the paper was relevant to human errors in software engineering. To eliminate false positive search hits, we considered the following cases:

   - **Generic Software Errors:** To eliminate papers discussing generic software errors (*i.e.* faults, failures, and/or defects), we specifically looked for mentions of human error theory or psychological concepts. If no mention of human error theory or psychological concepts was present in a paper, we considered that paper irrelevant. If a paper's abstract mentioned *human factors*, we erred on the side of caution and continued onto the next step.
   - **Generic Software:** To eliminate papers discussing software outside the domain of software engineering (*e.g.* medical software, point-of-sale systems), we looked specifically for the strings *software engineering* and/or *software development*, as well as software engineering concepts. If no mention of software engineering was present in a paper, we considered that paper irrelevant.
   - **Non-Research Sources:** We rejected a handful of search hits that were not research papers or software engineering books (*e.g.* workshop descriptions, standards documents, and opinion pieces).

   Papers deemed relevant were selected for review. If we could not make an informed accept/deny decision based on the title and abstract of a paper alone, we erred on the side of caution (following Brereton *et al.* [50]) and continued onto the next step.

2. **Examine Keywords in Context:** If the title and abstract were not enough to determine relevance, we searched the body of the literature for the search strings outlined in Figure 4.2. For searchable PDFs and web pages, we used the built-in search functionality for the PDF viewer and web browser, respectively. For non-searchable PDFs and physical papers/books, we manually examined the body text for keywords. We examined the keywords in context to determine whether or not the literature was discussing human errors in software engineering. Some examples:

   - If we got a search hit for "human error", but the paper only briefly mentioned human error during

Domain Strings	**Scope Strings**		
software engineering	human error	omission	SRK
software engineers	slip	omissions	skill-rule-knowledge
software development	slips	commission	James Reason
programmers	lapse	commissions	Jens Rasmussen
computing	lapses	skill errors	generic error modeling system
	mistake	rule errors	
	mistakes	knowledge errors	

Figure 4.2: List of Domain and Scope Strings

We used a combination of domain and scope strings to form search queries.

```
("software engineering" OR "software engineers" OR "software development" OR
  "programmers" OR "computing")
AND
("human error" OR "slip" OR "slips" OR "lapse" OR "lapses" OR "mistake" OR "mistakes"
  OR "omission" OR "omissions" OR "commission" OR "commissions" OR "skill errors" OR
  "rule errors" OR "knowledge errors" OR "SRK" OR "skill-rule-knowledge" OR
  "generic error modeling system" OR "James Reason" OR "Jens Rasmussen")
```

Figure 4.3: Boolean Logic Query used for IEEE Xplore and the ACM Digital Library

For IEEE Xplore and the ACM Digital Library, querying for the occurrence of any domain string *and* any scope string yielded good results.

the introduction/background/motivation and did not have results grounded in human error in software engineering, we considered that paper irrelevant.

- If we got a search hit for "lapse", but the paper only mentioned *time lapses*, we considered that paper irrelevant.
- If we got a search hit for "commission" in the copyright text of a paper, and no other mention of human error occurred, we considered that paper irrelevant.
- If we got a search hit for "rule-based", but the term was used in the context of rule-based classifiers, we considered that paper irrelevant.

3. **Review Related Work:** We repeated the above two steps for references in the related works sections of relevant papers to identify more relevant studies.

We repeated this process for each search hit until we encountered 40 search hits in a row that were deemed irrelevant or had already been found in a previous search engine. We used spreadsheets to keep track of search hits, their relevancy, and seen status. We reviewed a total of 284 studies, of which 16 were deemed relevant. Reviewing related work for relevant papers resulted in 52 more relevant studies, for a total of 68 studies related to human error in software engineering. Figure 4.5 visually summarizes our literature selection.

4.1.2.2 Data Extraction & Literature Quality

We read each paper deemed relevant to human errors in software engineering and took notes before summarizing each paper in Section 3.1. For each study, we examined:

- **Methodology:** What experiment(s) were conducted, who were the participants, and how was any collected data analyzed?
- **Results:** What human errors in software engineering were identified and documented in the literature? Were those human errors discussed in the context of a human error theory?
- **Conclusions:** What, if any, interesting findings or conclusions were presented?

```
software engineering human error OR software engineering slips lapses mistakes OR
software engineering skill rule knowledge OR software engineering omission commission
```

Figure 4.4: Boolean Logic Query used for Google Scholar

Google Scholar's search engine yielded many false positives with the query from Figure 4.3, so we used a more targeted search query. Quotes were not used because they changed how Google Scholar parses the query.

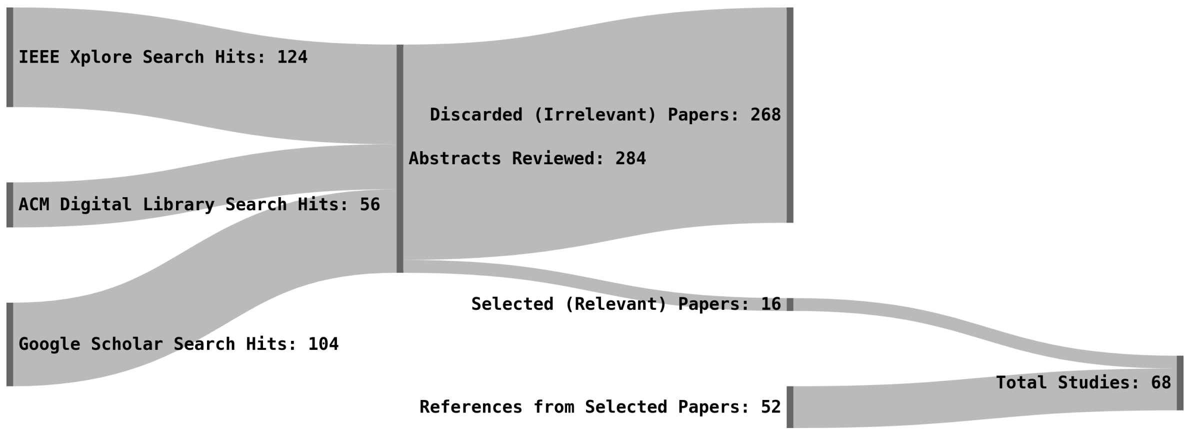


Figure 4.5: Visual Summary of Systematic Literature Review Results

Sankey diagram visually summarizing the results of our systematic literature review protocol (Section 4.1.2.1).

We took particular interest in papers that presented a taxonomy, or were themselves systematic literature reviews. After summarizing each study, we assessed the *quality* and *coverage* of each study.

Quality: Inspired by the properties of a high quality taxonomy described in previous work [9, 43, 127, 136, 174, 188, 192], we assessed the strengths and weaknesses of each previous research study about human errors in software engineering. We define strengths and weaknesses in terms of the following:

- **Scope:** Human error categories should be general to all phases of software engineering. We assigned a value of `0` or `1` indicating whether the human error categories in the study are general to all phases of software engineering (`1`) or not (`0`). For example, if a paper specifically discussed human errors during a single phase of software engineering (*e.g.* the work of Anu *et al.* [12, 15, 18, 22, 23] focusing only on requirements engineering), that paper received a scope score of `0`.

- **Ambiguity:** Human error categories and their definitions should be clear and unambiguous. We assigned a value of `0`, `1`, or `2` indicating whether a study's human error categories were completely ambiguous (`0`), somewhat ambiguous (`1`), or completely unambiguous (`2`). For example, if a paper only presented human errors in terms of a human error theory without providing specific human errors (*e.g.* Im & Baskerville [31, 147]), that paper received an ambiguity score of `0`. If some of the specific human errors presented in a paper were unclear (*e.g.* "expression errors" in Huang *et al.* [143]), that paper received an ambiguity score of `1`.

- **Completeness:** Human error categories should be exhaustive; edge cases should fit within an existing category. We assigned a value of `0` or `1` to each study, indicating whether we could imagine an edge case that does not fit within a study's human error categories (`0`) or not (`1`). For example, the taxonomy presented in Zapf *et al.* [370] does not have a category that time management errors could fit into, so this paper received a completeness score of `0`.

We then aggregated the scores for scope, ambiguity, and completeness into an overall quality score on a scale of `0` to `4`, with `4` being the highest quality and `0` the lowest. Studies that did not provide specific categories of human error (beyond the high level categories of slip/lapse/mistake, skill/rule/knowledge, and/or omission/commission) were automatically assigned a quality score of `0`.

Usefulness: We also evaluated the usefulness of each study based on their coverage of human error theories from cognitive psychology and their coverage of human errors in software engineering.

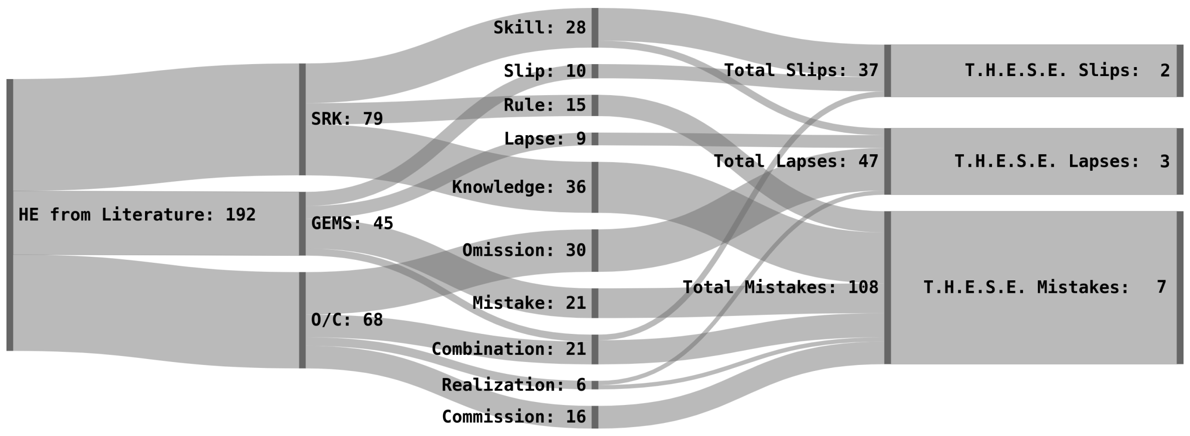


Figure 4.6: Visual Summary of T.H.E.S.E. Creation

Sankey diagram visually summarizing the creation of T.H.E.S.E. (Section 4.1.2.3). Key: HE—Human Error.

Table 4.1: Summary of Software Engineers' Human Errors from Literature

SRK Model		**GEMS Framework**		**Omission/Commission**	
Type	**Frequency**	**Type**	**Frequency**	**Type**	**Frequency**
Skill-Based	28	**Slip**	10	**Omission**	30
Rule-Based	15	**Lapse**	9	**Commission**	16
Knowledge-Based	36	**Mistake**	21	*Realization*	6
		Combination	5	*Combination*	16
Total	79	**Total**	45	**Total**	68

- **Coverage of Human Error Theories:** For each human error theory (GEMS Framework, SRK Model, Omission/Commission), we assigned a value of 0, 1, or 2 indicating whether a study's human error categories did not cover the human error theory at all (0), only partially covered the human error theory (1), or fully covered the human error theory (2). For example, Huang *et al.* [143] has human error categories spanning slips and lapses (but not mistakes), and knowledge (but not skill or rule) errors, so that paper received theory coverage scores of 1, 1, and 0 for Reason's GEMS framework, Rasmussen's SRK model, and errors of omission/commission, respectively.

- **Coverage of Human Errors in Software Engineering:** We mapped the human error categories from each study to the categories in T.H.E.S.E. For each category in T.H.E.S.E. that overlaps with categories in the study, the study received one point (maximum of 12).

An aggregate usefulness score was calculated as the sum of both coverage scores. Usefulness ranges from 0 to 18, with 18 being the most useful and 0 the least useful.

4.1.2.3 Creation of T.H.E.S.E.

To create our taxonomy, we needed to examine all of the human errors from literature and group them into categories of related human error. Following established good practices for taxonomy development from previous research [9, 43, 127, 136, 174, 188, 192], we wanted to avoid having too many categories while ensuring that the categories included were mutually exclusive and well-defined. We used the following process to create categories:

1. **Extract Human Errors:** We extracted specific human error categories (and their descriptions) from relevant papers into a list.
2. **Group Based on Theory:** Each human error theory groups human errors differently. Human errors discussed in the context of Rasmussen's SRK model were grouped into skill-, rule-, and knowledge-based categories;

Table 4.2: Summary of SLR Quality Scores

Study	**Scope (0-1)**	**Ambiguity (0-2)**	**Completeness (0-1)**	**Total Quality (0-4)**
[142, 253, 370]	1	2	0	3
[345]	1	1	0	2
[168]	1	1	0	2
[143, 161]	1	1	0	2
[20, 187]	0	2	0	2
[115]	0	2	0	2
[165]	0	2	0	2
[279]	0	2	0	2
Studies Involving HET*	0	2	0	2
[366]	0	1	0	1
Studies w/o Specific Categories†	0	0	0	0

* [12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 23, 138, 139, 140, 141, 194, 349]

† [3, 29, 30, 31, 32, 39, 64, 67, 68, 144, 147, 148, 171, 185, 186, 200, 252, 260, 273, 309, 315, 316, 317, 319]

human errors discussed in the context of Reason's GEMS framework were grouped into slip, lapse, and mistake categories; and human errors discussed in terms of omission and commission were grouped as such.

3. **Group Within Theory Categories:** Human errors within each category (*e.g.* slip, skill-based, omission) were compared with each other to determine whether or not they were the same human error, different human errors, or part of a subcategory of human error.
4. **Group Into Slips, Lapses, and Mistakes:** Since research indicates that the GEMS framework is consistent with how network administrators and security specialists view human errors [171], we created categories in T.H.E.S.E. that fall under slips, lapses, and mistakes.
   (a) **SRK to GEMS:** Since both rule- and knowledge-based human errors are considered mistakes, we placed those errors into the mistake category. Skill-based errors can be slips or lapses, so we placed them into one of those categories based on whether the skill-based human error was due to an attentional failure (slip) or a memory failure (lapse).
   (b) **Omission/Commission to GEMS:** Omissions are strictly lapses, so we placed them as such. Commissions can be slips or mistakes, so we placed them into the slip or mistake category based on their cause (attentional or planning failure, respectively).
5. **Repeat Step 3:** With skill-, rule-, and knowledge-based errors and errors of omission/commission grouped into their respective GEMS categories, we repeated Step 3 to eliminate any remaining duplicates.

For example, *overlooking design documentation* [20] (slip) and *overlooking or failure to read API documentation* [20] (slip) were grouped into a subcategory of human error, `Overlooking Documented Information (LS2)`. As another example, *inappropriate tool selected* [345] (rule-based mistake) and *problem solution errors* [15] (mistake) were considered the same category of human error, `Solution Choice Errors (LM6)`.

4.1.3 Results

In this section, we discuss our qualitative and quantitative results for **RQ1-3**. Note that while we reviewed 68 studies, some [276, 277, 353, 355, 356] were not included in our analysis for **RQ2** and **RQ3** because they did not fit the scope of software engineering. Additionally, we grouped all studies involving the Human Error Taxonomy (HET) [12, 14, 15, 16, 17, 18, 19, 21, 23, 138, 139, 140, 141, 194, 349] together for our analysis. Percentages reported for **RQ2** and **RQ3** are based on a total of 49 studies.

Table 4.3: Summary of SLR T.H.E.S.E. Coverage

Study	**LS1**	**LS2**	**LL1**	**LL2**	**LL3**	**LM1**	**LM2**	**LM3**	**LM4**	**LM5**	**LM6**	**LM7**	**Total (0-12)**
Studies Involving HET*	1	1	0	0	0	0	1	1	1	1	1	0	7
[20]	0	1	1	1	1	0	1	1	0	0	1	0	7
[168]	1	0	1	0	0	1	1	1	0	0	1	0	6
[345]	0	0	0	0	0	0	1	1	1	1	1	0	5
[15]	1	1	0	0	0	0	0	1	1	0	1	0	5
[370]	0	1	0	0	0	0	1	1	0	0	1	0	4
[161]	1	1	0	0	0	0	0	1	0	0	0	1	4
[143]	0	0	0	0	0	0	1	1	0	0	0	1	3
[142]	1	0	0	0	0	1	0	0	0	0	0	0	2
[115, 187, 279]	0	0	0	0	0	1	0	0	0	0	0	0	1
[366]	0	0	0	0	0	0	1	0	0	0	0	0	1
[165, 253]	0	0	0	0	0	0	0	0	0	0	0	0	0
Studies w/o Specific Categories†	—	—	—	—	—	—	—	—	—	—	—	—	0

* [12, 13, 14, 16, 17, 18, 19, 21, 22, 23, 138, 139, 140, 141, 194, 349]

† [13, 22, 29, 30, 32, 64, 67, 68, 148, 165, 171, 185, 186, 200, 252, 253, 260, 273, 309, 315, 316, 317, 319, 372]

4.1.3.1 Human Error Discovery

RQ 1: What human errors experienced by software engineers are documented in previous research?

We identified 192 human errors in software engineering from 68 research studies. 79 human errors were based on Rasmussen's skill-, rule-, and knowledge-based human error theory, 45 were based on Reason's slips, lapses, and mistakes, and 68 were errors of omission or commission. Mistakes (both rule- and knowledge-based) were the most frequent type of human error discussed in literature, differing from results in domain-independent psychology research from Reason [298].

Our systematic literature review of 68 studies about human errors in software engineering revealed 192 human errors spanning three theories of human error. Figure 4.5 visually summarizes the results of our systematic literature review protocol. Table 4.1 summarizes the number of each type of human error. Specific categories of human error are outlined in Section 3.1. We found that the majority of human errors discussed in previous literature are mistakes (mistakes, skill-based errors, and rule-based errors). This finding is inconsistent with previous findings from Reason [298]. However, we believe this finding makes sense in the context of software engineering; mistakes are planning failures and software engineering is a highly plan-oriented domain.

After reviewing literature for human errors in software engineering, we grouped related human errors together into a Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.) following the process in Section 4.1.2.3. Our taxonomy (Table 4.6) has 12 categories of human error spanning slips, lapses, and mistakes. Figure 4.6 visually summarizes the creation of T.H.E.S.E. We chose to use slips, lapses, and mistakes as the basis for our taxonomy following Kraemer & Carayon's finding that Reason's GEMS framework is consistent with how network administrators and security specialists view human errors [171]. Additionally, Rasmussen's SRK model and errors of omission and commission overlap with Reason's GEMS framework. The majority of categories in T.H.E.S.E. are mistakes, mirroring the frequency of mistake categories compared to slips and lapses found during our systematic literature review.

Interestingly, the human errors in software engineering studies that we examined did not include time management errors or inadequate testing. The absence of these colloquially common human errors suggests that existing research about human errors in software engineering is incomplete; notably that none of the 68 studies we reviewed examined software engineers' human errors in the wild (*e.g.* bug reports, vulnerability disclosures).

Table 4.4: Summary of SLR Usefulness Scores

Study	**Theory Coverage (0-6)**	**T.H.E.S.E. Coverage (0-12)**	**Total (0-18)**
[168]	3	6	9
Studies Involving HET*	2	7	9
[20]	2	6	8
[161, 370]	3	4	7
[15, 345]	2	5	7
[143]	2	3	5
[3]	4	0	4
[144, 147]	3	0	3
[142]	1	2	3
Studies w/o T.H.E.S.E. Coverage†	2	0	2
[115, 187, 279, 366]	1	1	2
[39]	1	0	1
[31]	0	0	0

* [12, 14, 16, 17, 18, 19, 21, 23, 138, 139, 140, 141, 194, 349]

† [13, 22, 29, 30, 32, 64, 67, 68, 148, 165, 171, 185, 186, 200, 252, 253, 260, 273, 309, 315, 316, 317, 319, 372]

Table 4.5: Summary of SLR Human Error Theory Coverage

Study	**GEMS**	**SRK**	**OC**	**Total (0-6)**
[3]	2	2	0	4
[161, 168, 370]	0	2	1	3
[144, 147]	1	2	0	3
[253]	1	1	1	3
[143, 309]	1	1	0	2
[12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 138, 139, 140, 141, 171, 194, 252, 349]	2	0	0	2
[64, 67, 68, 185, 186, 345, 372]	0	2	0	2
[29, 30, 32, 148, 165, 200, 260, 273, 315, 316, 317, 319]	0	0	2	2
[39, 142, 366]	0	1	0	1
[115, 187, 279]	0	0	1	1
[31]	0	0	0	0

4.1.3.2 Human Error Appraisal

RQ 2: What are the strengths and weaknesses of previous research about human error in software engineering? We define strengths and weaknesses in terms of software engineering scope, category ambiguity, and category completeness.

Nearly half (49%) of studies did not include specific human error categories, 59% of studies contained some level of ambiguity in categories and their descriptions, and 0% of studies were able to accommodate all potential human errors experienced by software engineers. Only 15% of the literature we reviewed had a scope general to all software engineering phases.

As described in Section 4.1.2.2, we assessed the quality of previous research studies' human error categories based on three properties of a high quality taxonomy: software engineering scope, human error category ambiguity, and human error category completeness. Each of those properties was assigned a numerical score, and scores were summed for each study to yield an overall human error quality score. Table 4.2 summarizes these scores.

We found that 49% (24/49) of the research studies about human error in software engineering did not report specific human error categories beyond the high level slip/lapse/mistake, skill/rule/knowledge, and omission/commission categories. This is concerning because without specific categories, replications of these studies cannot compare results. Further, the lack of specific human error categories limits the insights that software engineers can glean from these studies.

59% (29/49) of the research studies that we reviewed contained some level of ambiguity within human error categories and their descriptions. Software engineers will find that categorizing their own human errors will be much more difficult when the definitions are ambiguous.

Only 15% (7/49) of the studies we reviewed had a scope general to all phases of software engineering. The other 85% of studies had narrow scopes, primarily focused on requirements engineering and implementation. While the human error categories from these studies may be useful in specific contexts, software engineers are involved in all phases of software engineering and thus need to be able to categorize and confront the human errors that they experience during every phase.

None of the studies we reviewed had human error categories capable of accommodating all potential human errors that software engineers could experience. To reach this conclusion, we closely examined T.H.E.S.E. and proposed a new category of human error: errors related to time management. We reexamined all of the studies and found that none of them considered time management errors.

These results indicate that we do not yet have a complete understanding of human errors in software engineering, but previous research has laid the groundwork for T.H.E.S.E., and in Section 4.2, we explored one of the gaps present in existing research.

4.1.3.3 Human Error Usefulness

RQ 3: How comprehensive is existing research about human errors in software engineering? We define usefulness in terms of coverage of human error theories from cognitive psychology and coverage of human errors in software engineering activities.

Of the human error studies, the highest usefulness score was 9 out of 18. Coverage of human error theories varied; no study fully covered all three human error theories, 16% of studies partially covered two human error theories, and 84% of studies covered only one human error theory. Coverage of human errors in software engineering was also poor, with 53% of studies having no coverage.

As described in Section 4.1.2.2, we assessed the usefulness of previous research studies' human error categories based on both their coverage of human error theories and their coverage of human errors in software engineering. Each human error theory and each category in T.H.E.S.E. was assigned a numerical score, and scores were summed for each study, yielding an overall human error usefulness score. Table 4.4 summarizes total human error usefulness for each study. Table 4.3 and Table 4.5 summarize coverage of human errors in software engineering and coverage of human error theories, respectively.

We found that 4% (2/49) of studies about human error in software engineering had an overall usefulness score of 50%. All other studies had a usefulness score less than 50%. This reveals that individual studies about human errors in software engineering are not particularly useful on their own. Only one study that we reviewed partially covered all three human error theories (no studies fully covered all three human error theories) and 53% (26/49) of studies had no overlap with T.H.E.S.E. (because these studies did not describe the specific human errors experienced by developers).

16% (8/49) of studies partially covered two human error theories, typically a combination of Reason's GEMS framework and Rasmussen's SRK model. The other 84% of studies only (fully or partially) covered a single human error theory. This reveals that the majority of reviewed research about human errors in software engineering only considered more narrowly focused aspects of human error. As a developer confronts their human errors, knowing only that a human error they have experienced is a mistake does not provide a full understanding. Was the human error a rule-based or knowledge-based mistake? Knowing only that the human error experienced was skill-based does not reveal whether the human error was due to an attentional (slip) or memory (lapse) failure. To prevent human errors in the future, software engineers need a complete understanding of their human errors.

The finding that 53% of studies that we reviewed had no overlap with T.H.E.S.E. is because those studies did not reveal the specific human error categories beyond the high level categories in each human error theory. As discussed in Section 4.1.3.2, without specific categories, these studies cannot be exactly replicated, and the insights that software engineers can gain from them are limited.

Of the 47% of studies that *do* have some overlap with T.H.E.S.E., most did not consider human errors related to code complexity (`LM7`), communication with project stakeholders/users (`LM5`), or code logic (`LM1`). Additionally,

Table 4.6: Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.) Version 1

ID	Category/Definition
Slips	
LS1	**Syntax Errors:** Any error in coding language syntax that impacts the executability of the code. Note that Logical Errors (*e.g.* `+=` instead of `+`) are not Syntax Errors.
LS2	**Overlooking Documented Information:** Errors resulting from overlooking documented information, such as project descriptions, stakeholder requirements, API/library/tool/framework documentation, coding standards, programming language specifications, and bug/issue reports.
Lapses	
LL1	**Forgetting to Fix a Defect:** Forgetting to fix a defect that you encountered, but chose not to fix right away.
LL2	**Forgetting to Remove Development Artifacts:** Forgetting to remove debug log files, dead code, informal test code, commented out code, test databases, backdoors, *etc.*
LL3	**Forgetting to Save Work:** Forgetting to push code, or forgetting to backup/save data or documentation.
Mistakes	
LM1	**Code Logic Errors:** A code logic error is one in which the code executes, but produces an incorrect output/behavior due to incorrect logic. Examples include using incorrect operators (*e.g.* `+=` instead of `+`), erroneous if/else statements, incorrect variable initializations, problems with variable scope, and omission of necessary logic.
LM2	**Incomplete Domain Knowledge:** Errors resulting from incomplete knowledge of the software system's target domain (*e.g.* banking, astrophysics).
LM3	**Wrong Assumption Errors:** Errors resulting from an incorrect assumption about system requirements, stakeholder expectations, project environments (*e.g.* coding languages and frameworks), library functionality, and program inputs.
LM4	**Internal Communication Errors:** Errors resulting from inadequate communication between development team members.
LM5	**External Communication Errors:** Errors resulting from inadequate communication with project stakeholders, third-party contractors, or users.
LM6	**Solution Choice Errors:** Misunderstood problem-solving methods/techniques result in analyzing the problem incorrectly and choosing the wrong solution. For example, choosing to implement a database system in Python rather than using SQL.
LM7	**Code Complexity Errors:** Errors resulting from misunderstood code due to poor documentation or unnecessary complexity. Examples include too many nested if/else statements or for-loops and poorly named variables/functions/classes/files.

most of these studies did not consider lapses at all. The categories of human error often considered in these studies included wrong assumption errors (`LM3`) and solution choice errors (`LM6`).

4.1.4 Limitations

Taxonomy developers should strive to meet certain criteria to ensure their taxonomies are beneficial. To be widely accepted and beneficial, our taxonomy needs to be of high quality. According to previous research [9, 43, 127, 136, 174, 188, 192], a good taxonomy should have the following properties:

- **Accepted:** Our taxonomy should be "logical and intuitive [136]" to ensure that it can be accepted by developers and researchers.
- **Complete/Exhaustive:** Any developer error should be able to be categorized using our taxonomy; there should be no edge cases that our taxonomy does not cover.

- **Comprehensible/Understandable:** Developers at all skill-levels, security professionals, and researchers should be able to easily understand our taxonomy.
- **Deterministic:** The process for classifying developer errors within our taxonomy should be clearly defined. We will ensure this by creating a methodology to use our taxonomy, similar to the goal of Anu *et al.*'s HEAA [14].
- **Mutually Exclusive:** Developers errors should be able to be categorized into one, and only one, class of human error (*i.e.* slip, lapse, or mistake), and one and only one specific category of human error.
- **Objective:** The classification process for our taxonomy should combat any subjectivity on the part of the person performing the classification.
- **Primitive:** Questions used to guide classification should have simple "yes" or "no" answers.
- **Repeatable:** Classification of developer errors into human error should be repeatable. Our classification methodology will also serve to meet this goal.
- **Unambiguous:** Each class of human error (*i.e.* slips, lapses, and mistakes) in our taxonomy should be well-defined to ensure no ambiguity in the classification process.
- **Useful:** Our taxonomy should be useful and provide a benefit to developers.
- **Well Defined Terminology:** All terms used in our taxonomy should be clearly defined to avoid confusion. Terminology should also be specific enough to avoid ambiguity, and outdated or not-yet-adopted terms should be avoided.

Since this was an exploratory systematic literature review, we cannot ensure that all these properties of a high quality taxonomy are met. Chapter 5 addresses some of these concerns.

Brereton *et al.* notes specifying research questions as the "most critical element of a systematic review [50]." We addressed this concern by forming our research questions before conducting our SLR. As Brereton *et al.* notes, revision of research questions is to be expected as understanding of the problem grows [50]; as we encountered more relevant studies, **RQ2** and **RQ3** evolved based on the measurable aspects of study quality we observed. Brereton *et al.* also recommends developing a detailed plan (*e.g.* a review protocol) to guide the SLR, noting that all members of the study need to be involved in designing the review protocol and that piloting the review protocol is essential. Our initial review protocol (not discussed previously in this work) was rather unsystematic; we queried Google Scholar with a not-previously-defined set of search strings and ill-defined stopping criteria. This pilot revealed that the literature we were looking for existed, but our methodology needed improvement, which prompted us to create defined search queries, stopping criteria, and other elements of a review protocol. Additionally, we used three different search engines (Google Scholar, IEEE Xplore, and ACM Digital Library) and modeled our review protocol following suggestions made by Brereton *et al.* [50].

4.1.5 Summary

In this section, we performed a systematic literature review of research about human errors in software engineering. We identified 192 human errors from 68 research studies and aggregated those human errors into 12 categories within a Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.). Categories in T.H.E.S.E. span slips (attention failures), lapses (memory failures), and mistakes (planning failures) from James Reason's Generic Error-Modelling Framework. Our key results are summarized as follows:

- The majority of human error categories discussed in software engineering literature are mistakes (in the GEMS sense of the word)
- 49% of studies do not detail specific human error categories beyond the high level categories of slip/lapse/mistake, skill/rule/knowledge, and omission/commission
- 59% of studies have human error categories (and descriptions) that are (at least partially) ambiguous
- 85% of studies have a scope too narrow to cover all software engineering activities
- 84% of studies only consider one human error theory

apology, *n.*: "justification, explanation, or excuse, of an incident or course of action [269]."

Figure 4.7: Oxford English Dictionary Apology Definition
This apology definition is used throughout Chapter 4.2.

Our systematic literature review outlines an almost 40-year history of concern about human error in software engineering. Despite all of that effort and the insights from 68 studies, the software engineering domain still lacks an established and accepted human error assessment process, a process which could improve the quality and security of software, as it has done in the medical [98, 159, 305] and transportation [300, 332] domains. Our findings reveal that this area of research is vast, and the effort to study human error in software engineering has only begun, with most studies being narrow in software engineering and human error scope, as well as lacking specific details about observed human errors. We do not intend to belittle these studies as they have been valuable in our ongoing effort ***to help software engineers confront and reflect on their human errors by creating a process to document, organize, and analyze human errors***. We envision T.H.E.S.E. as an integral step toward assessment. By confronting their human errors, software engineers can identify areas of improvement, and their managers can identify systemic human errors within development teams and implement process improvements in response. This study was our first step toward making human error assessment accessible to software engineers, which, we believe, will ultimately lead to improved quality and security stature of future software products.

4.2 Human Errors from Software Engineering Artifacts

4.2.1 Motivation & Research Questions

As we established in Section 4.1, software engineering is a complex process, and despite their best efforts, software engineers inevitably experience human errors. While software developers' human errors have been documented in controlled experiments and interviews, to our knowledge, developers' **self-admitted** human errors in development artifacts have not been studied. One way to identify (some of) developers' self-admitted human errors is to mine apologies—admissions of error [211]—in development artifacts. Throughout this work, we used the apology definition shown in Figure 4.7. Modern software engineering affords massive, rich records of socio-technical, natural language interaction in public venues, such as with open source software development on GitHub. Specifically, these records can provide insight into developers' **self-admitted** human errors, as evidenced by their apologies.

Our long-term aim is to help software engineers confront and reflect on their human errors with a formal **human error assessment** process, but before we can do so, we need to understand how human error theories from cognitive psychology manifest themselves in software engineering artifacts. In this study, we addressed the following research questions:

RQ 4: **Identifying Developers' Apologies**
Can apology lemmas reliably identify developers' apologies in development artifacts?

RQ 5: **Anatomy of Developers' Apologies**
How often do developers apologize and which apology lemmas are most common?

RQ 6: **Developers' Self-Admitted Human Errors**
Which human errors from literature do developers admit to? Which human errors from developer apologies do not exist in literature?

4.2.2 Methodology

This section outlines our methodology for collecting, annotating, and analyzing a subset of software engineers' self-admitted human errors. The dataset we collected is available on Zenodo [214] and the code used throughout this study is available on GitHub [212].

4.2.2.1 Data Collection & Preprocessing

We used GitHub's GraphQL API [112] to download developers' comments from 17,378 repositories spanning 45 programming languages. We chose repositories that were open source and widely used (*i.e.* at least 850 GitHub stars), and had a primary language that is commonly used. We define *commonly used* languages as those appearing in the top 50 of the TIOBE Index [340] and/or those marked *popular* in GitHub's advanced search page [109] as of August 31, 2021. These languages are outlined in Figure 4.8.

Table 4.7: Summary of GitHub Data Collected

	Number	**Comments**
Commits	49,710,108	979,642
Issues	13,458,208	54,252,380
Pull Requests	12,680,373	33,408,215
Total	—	88,640,237

ABAP	COBOL	Fortran	LabVIEW	Perl	SAS	TSQL
Ada	CoffeeScript	Go	Lisp	PHP	Scala	TypeScript
Apex	CSS	Groovy	Logos	PowerShell	Scheme	VBScript
Assembly	D	HTML	Lua	Prolog	Scratch	VHDL
C	Dart	Java	MATLAB	Python	Shell	VB .NET
C#	DM	JavaScript	Nim	R	SQL	
C++	Elixir	Julia	Objective-C	Ruby	SQLPL	
Clojure	F#	Kotlin	Pascal	Rust	Swift	

Figure 4.8: Commonly Used Programming Languages

As of August 31, 2021, these are the top 50 commonly used programming languages as reported by the TIOBE Index [340].

We queried GitHub's GraphQL API for repositories whose primary language is one of the commonly used languages in Figure 4.8. GitHub's GraphQL API returns at most 1000 repositories for each target language. We then downloaded developers comments on commits, issues, and pull requests for each matching repository using GitHub's GraphQL API. We downloaded the title, ID number, author, creation date, URL, and raw description text for each commit, issue, and pull request. For each comment, we collected the raw text, author, creation date, and URL. Data received from GraphQL queries was converted from JSON into CSV before saving to disk.

Of the 17,491 repositories matching our criteria, 113 could not be accessed due to various GraphQL API errors. Seven target languages—Ada, Apex, LabVIEW, Logos, SAS, SQL, and SQLPL—had no repositories matching the above criteria, and no repositories for VB .NET could be accessed due to the aforementioned API errors. In total, we collected over 88.6 million developer comments, as summarized in Table 4.7.

We preprocessed the raw developer comments from our GitHub dataset by lowercasing all text, removing punctuation (*i.e.* periods, commas, colons, semicolons, question marks, and exclamation points), and lemmatizing. Lemmatizing reduces grammatically related words into *lemmas*, simpler common base forms, using the word's context (the words surrounding it) and morphology [196]. For example, the words *"bug"*, *"bugs"*, and *"bug's"* all have the same lemma, *bug*. We used regular expressions to remove punctuation and the natural language processing library SpaCy [243] to lemmatize developer comments.

4.2.2.2 Apology Annotation

A common way for human errors to be apparent in natural language is through apologies. We used the Oxford English Dictionary's definition of an apology (Figure 4.7) throughout this work. We reviewed 12 linguistics research papers about apologies and documented the common apology lemmas in Table 4.8, along with example apology statements from our dataset. In addition to the 12 apology lemmas identified from linguistics literature, we also considered the lemmas *apology*, *mistake*, and *mistaken* to be indicative of an apology (*e.g. My apologies*; *that's my mistake*; *I was mistaken*). We automatically classified developer comments containing at least one apology lemma as apologies.

Two researchers—both native English speakers from the United States and graduate computer science students—independently annotated a random subset of 1,237 developer comments to verify whether developer comments containing at least one apology lemma were apologies. We used `scikit-learn` [282] to compute inter-annotator agreement (Cohen's κ). All disagreements between annotators were discussed until resolved.

We implemented a naïve *classifier* that simply counts the number of apology lemmas present in a developer comment. If one or more apology lemmas are present, our classifier assigns a label of '1' indicating that the developer comment contains an apology, or '0' if no apology lemmas are present. We report results for this initial classifier with the label `v1`. After completing our classifier, we manually examined false positives (*i.e.* comments automatically classified as apologies when they should not have been). Our manual examination yielded a set of lemma phrases that result in false positive classifications. We updated our classifier with special rules to handle common false positive

Table 4.8: Apology Lemmas Identified from Linguistics Literature

Lemma	Sources	Example Apology from Our Dataset
admit	[178, 346]	*I admit I missed that part of the doc so it's definitely not a bug.*
afraid	[36, 133]	*I'm afraid I can't really help debugging it.*
apologize/se	[36, 65, 172, 178, 206, 311, 346]	*I apologize for leaving this issue hanging.*
blame	[133, 346]	*I think I'm to blame for this.*
excuse	[36, 47, 65, 133, 172, 206, 268, 346]	*Excuse my beginner's incompetence.*
fault	[133, 311, 346, 365]	*Yes that's my fault.*
forgive	[36, 65, 133, 172, 178, 206, 346]	*Please forgive my ambiguity.*
forgot	[133, 268]	*I forgot to evaluate using multiple metrics.*
oops	[133]	*Oops looks like I had connection_fileds instead of connect_fields.*
pardon	[36, 133, 206, 311, 346]	*Please pardon my misunderstanding.*
regret	[36, 133, 178, 268, 311]	*I regret that we didn't manage to add this back at the time.*
sorry	[36, 47, 65, 133, 172, 178, 206, 268, 311, 346, 365]	*I'm sorry for the terribly delayed response.*

lemma phrases (outlined in Table 4.9). We report results for this updated classifier with the label `v2`. Upon closer inspection, we identified three apology lemmas—"admit", "afraid", and "forgot"—that were missing from our classifier. We added these apology lemmas (with relevant false positive lemma phrases) to the final version of our classifier, labeled `v3`.

We also evaluated our classifier against manual dialog act annotations in the Switchboard corpus [116] of spontaneous telephone conversations. The Switchboard corpus is included in popular natural language processing libraries—such as the Cornell Conversational Analysis Toolkit [60] and the Natural Language Toolkit [38]—and has over 10,000 and 36,000 search hits on Google Scholar and GitHub, respectively. The 221,616 utterances in the Switchboard corpus are annotated with 43 dialog act labels (*e.g.* apology, yes-no-question, statement-opinion). 79 (0.0004%) utterances are labeled as apologies. We used the SWDA python code [156, 287, 325, 333] to access the Switchboard corpus and accompanying dialog act annotations.

4.2.2.3 Human Error Categorization

In Section 4.1, we examined 68 research papers about human error in software engineering to create a Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.). Our literature review yielded 12 categories of human error in software engineering, spanning slips, lapses, and mistakes. These categories are outlined in Table 4.6. Using these categories, we manually annotated a random subset of 332 apology comments, labeling each with a specific human error (from T.H.E.S.E.). We used the following process to categorize apology comments as human errors. Our process is also summarized visually in Figure 4.9.

Step 1: **Evaluate Comment:** Read the comment text, the comment author's name, the comment author's description, and the comment author's role.

- **Evaluate Apology:** Determine if the author's comment is apologizing or if the apology lemma(s) present are a false positive (*e.g.* apology lemma is part of quoted text; "It's ok, I'm not blaming you."). If the apology lemma(s) present are a false positive, do not continue with categorization.
- **Disregard Bots:** If the comment author is a bot (*e.g.* 'ti-srebot'), do not continue with categorization. We manually examined author names and related metadata to determine if a comment author was a bot. For example, we identified the GitHub user 'dependabot' as a bot, since the word 'bot' is in the username and the accompanying description indicates that it is an automated tool.
- **Identify Author Role:** If the comment author is not a bot, determine whether they are a developer or a user based on role labels (*e.g.* `Contributor`, `Author`, `Member`, `Collaborator`, `Owner`, or no label) and context. If the comment author is a user, see the special case in Step 3.

Table 4.9: False Positive Apology Lemma Phrases

Lemma Phrase	Example from Our Dataset
not afraid	*But as you are **not afraid** to tinker, let's see if I can just point you in the right direction.*
n't afraid	*You can change whatever you want if you are**n't afraid** of getting dirty with the dlib code.*
not apologize/se	*You should **not apologize** for the bug report, just the opposite.*
n't apologize/se	*Ca**n't apologize** enough for the delay–hate to see that this one slipped through the cracks.*
git blame	*Pointing fingers is best left to **git blame**.*
n't blame	*Doing that consistently... would have been tricky, so I ca**n't blame** the team for that!*
not to blame	*Turns out it was a deadlock problem after all, but SciPy is **not to blame**.*
seg fault	*I was also getting **seg fault**s due to timeouts of this coroutine.*
segmentation fault	*I tried compiling with 3.0.0a8 and there was no **segmentation fault**.*
page fault	*Map will trigger **page fault**s, which can be very inefficient.*
permission fault	*It's obviously a **permission fault**.*
protection fault	*It's more like a general **protection fault**.*
not <PRONOUN> fault	*This is crazy (and obviously **not your fault**).*
not a mistake	*It's likely **not a mistake** on your part.*
not mistaken	*If I'm **not mistaken**, the socket should be in blocking mode.*
n't regret	*Anyway, I do**n't regret** reverting this changeset.*
better safe than sorry	*No idea whether this is even worth reporting, but **better safe than sorry**.*
not sorry	*We are **not sorry** for any of the actions on the pull request or the issue thread.*

Step 2: **Categorize Slip/Lapse/Mistake:** Determine whether the human error being discussed is a slip, lapse, or mistake.

- **Ambiguous Human Errors:** If there is not enough context to categorize a human error, do not continue with categorization. Note that developers categorizing their own human errors will have all of the necessary context to do so.

Step 3: **Categorize Human Error:** Choose the most accurate category for the human error being discussed.

- **User Apology:** If the apology comment author is a user, determine whether a communication error between the user and developers has occurred. If so, categorize as `External Communication Errors (LM5)`. If not, do not continue with categorization.
- **Ambiguous Human Errors:** If the human error doesn't quite fit into an existing category, continue to Step 4.

Step 4: **Disambiguate Human Errors:** Determine what to do about ambiguous human errors that don't quite fit into an existing category or human errors that are categorized as a general slip, lapse, or mistake.

- **Imperfect Fits:** If the human error doesn't quite fit into an existing category, but could fit if the category definition is slightly modified (and can be modified without invalidating previous categorizations), update the category definition.
- **General Slip/Lapse/Mistake:** If the human error is a general slip, lapse, or mistake and does not fit in an existing category, define a new category if necessary. Any potential new categories were discussed with other researchers before being finalized.

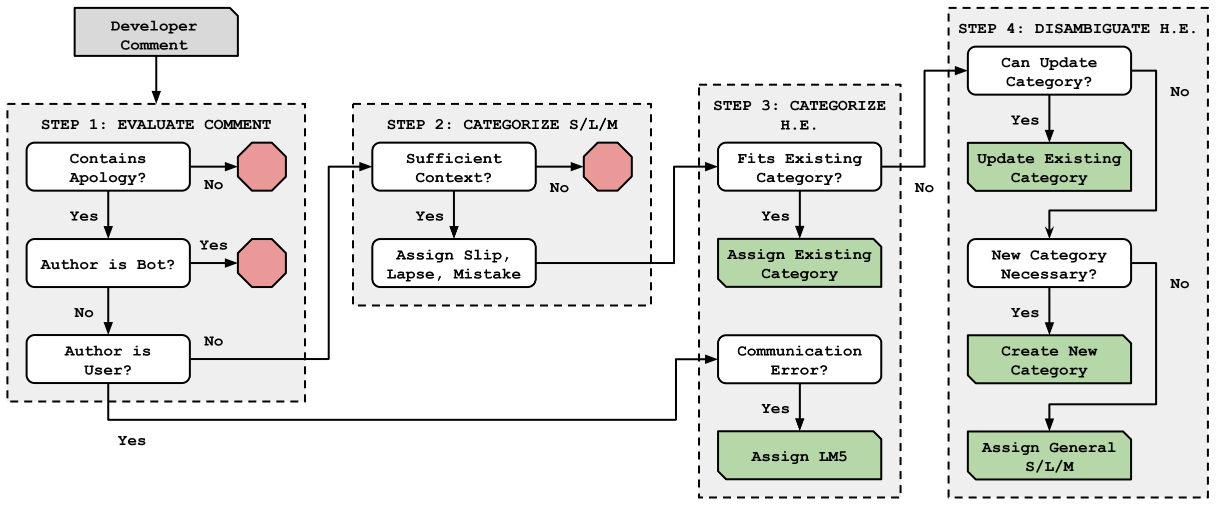


Figure 4.9: Visual Summary of Our Manual Human Error Categorization Process

Key: H.E.—Human Error; S/L/M—Slip, Lapse, or Mistake; Red Octagon—Do Not Continue with Categorization. `External Communication Errors (LM5)` and other human error categories are outlined in Table 4.6 and Table 4.16.

Table 4.10: Automatic Apology Classification Performance on Manually Annotated Apologies

Classifier	# Samples	Precison	Recall	F1	Accuracy
v1	1,237	0.417	0.997	0.829	0.879
v2	1,237	0.417	0.997	0.869	0.911
v3	1,237	0.417	0.997	0.869	0.911

To reach a reasonable level of convergence, we repeated this process until we had 50 human error categorizations, at which point we counted how many new categories had been created. We continued in increments of 50 categorizations until we were able to categorize 50 human errors without creating a new category.

4.2.3 Results

In this section, we discuss our qualitative and quantitative results for **RQ4-6**.

4.2.3.1 Identifying Developers' Apologies

RQ 4: Can apology lemmas reliably identify developers' apologies in development artifacts?

91% of developer comments containing at least one apology lemma matched our manual apology annotation. Our approach to automatic apology classification achieved near perfect recall (99%) with a high F1 score (87%).

After automatically classifying developer comments as apologies based on apology lemmas (see Section 4.2.2.2), two researchers independently annotated 1,237 developer comments with almost perfect agreement (Cohen's κ = 0.94). 91% of developer comments containing at least one apology lemma matched our manual apology annotation. Our automatic classification process achieved near perfect recall with high accuracy and F1 score, as outlined in Table 4.10.

Of the 1,237 developer comments manually annotated, automatic apology annotation resulted in one false negative and 109 false positives. The false negative was due to a typo in the developer's original comment. The majority of false positives were due to the apology lemmas "mistake" (*e.g. I found a mistake*), "fault" (*e.g.* discussing a *faulty* package) and "sorry" (*e.g.* saying *sorry* to be polite). Other false positives were due to apology lemmas appearing in quoted text, URLs, or email signatures (*e.g. please excuse my brevity*).

Table 4.11: Automatic Apology Classification Performance on the Switchboard Corpus

Classifier	# Apologies	Precision	Recall	F1	Accuracy
v1	491 (0.0022%)	0.149	0.924	0.256	0.998
v2	469 (0.0021%)	0.156	0.924	0.266	0.998
v3	598 (0.0027%)	0.122	0.924	0.216	0.998

Table 4.12: Summary of Apology Comments in GitHub Dataset

		Apology Comments		
	Comments	**v1**	**v2**	**v3**
Commits	979,642	25,214 (2.57%)	24,710 (2.52%)	26,910 (2.75%)
Issues	54,252,380	1,698,186 (3.13%)	1,657,114 (3.05%)	1,793,460 (3.31%)
Pull Requests	33,408,215	915,607 (2.74%)	899,015 (2.69%)	971,476 (2.91%)
Total	88,640,237	2,639,007 (2.98%)	2,580,839 (2.91%)	2,791,846 (3.15%)

Our apology classification process labels 598 (0.0027%) utterances from the Switchboard corpus [116] as apologies, with very high recall (0.92) and accuracy (0.99). Table 4.11 summarizes our classification results. While our apology lemma based approach to classifying apologies yields different results from manual annotation on the Switchboard corpus, the high recall indicates that our approach is very good at identifying apologies with few false negatives, but tends to over-predict (*e.g.* low precision). This discrepancy is likely due to human annotators using conversational context to annotate apologies in the Switchboard corpus; our automatic apology classification process does not use conversational context.

The majority of false positives in the Switchboard corpus included the apology lemmas *sorry*, *excuse*, *admit*, or *mistake*. While some of these false positives are just that (*e.g.* "it makes you feel really sorry for her [116]"; "It was a tragic mistake [116]"), others could be considered apologies. For example, speakers often said "I'm sorry [116]" or "excuse me [116]" after coughing/sneezing or when they did not hear something that was said—these would be considered apologies according to Kramer & Moore [172] and the definition used throughout this work (see Figure 4.7).

4.2.3.2 Anatomy of Developers' Apologies

RQ 5: How often do developers apologize and which apology lemmas are most common?

Software engineers apologize to each other on GitHub frequently, with an apology lemma density over one thousand times greater than speakers in the Switchboard corpus. On average, developers' apologies are 36 words (67%) longer than other developer comments. "Sorry" is the most common apology lemma, followed by "mistake", "fault", "apology", and "afraid."

Of the 88.6 million developer comments we collected, 2.7 million (about 3%)[4] contained at least one apology lemma. Ordinarily, apologies are relatively rare in human conversations, occurring in less than 0.0004% of utterances in the popular Switchboard corpus [116] of telephone conversations. We suspect that this higher propensity to apologize among software developers[5] is likely due to the impact of human errors on peer developers, project stakeholders, and users. Since apologies are an aspect of politeness in conversations [53], frequent apologies made by developers may be an attempt to maintain positive working relationships. A breakdown of apology comments for commits, issues, and pull requests is provided in Table 4.12.

Developers' apology comments are 36 words (67%) longer than their other comments on average (Table 4.13). Human errors in software engineering have a lasting impact, so this result makes sense—to improve software, developers' apologies need to go beyond admitting that something went wrong, they should explain what went wrong and how it can be addressed. An example of one such apology is included in Figure 4.10.

[4]If we consider the 8.8% false positive rate from manual apology annotation (see Section 4.2.3.1), then the total number of apology comments in our dataset drops to about 2.5 million comments (2.9%)—which is still an apology lemma density over one thousand times greater than in the Switchboard corpus.

[5]We must note that direct comparison with the Switchboard corpus is dubious given the difference in annotation schemes and conversational context; the different rate of apologies among software engineers is intriguing and should be explored in future work.

Table 4.13: Word Count Statistics for Developer Comments (Classifier v3)

		Word Count			
	# Comments	**Average**	**Median**	**Minimum**	**Maximum**
Apology	2,791,846	91.32	40	1	130,140
Non-Apology	85,848,391	54.68	22	1	257,291

Table 4.14: Apology Lemma Frequency in Developer Comments

Lemma	**Frequency**	**Lemma**	**Frequency**	**Lemma**	**Frequency**
admit	48,447	**excuse**	33,645	**mistaken**	24,467
afraid	127,040	**fault**	141,156	**oops**	69,965
apology	137,040	**forgive**	16,957	**pardon**	8,331
apologize/se	64,111	**forgot**	65,008	**regret**	7,502
blame	60,616	**mistake**	291,384	**sorry**	1,923,163

Table 4.14 lists the frequencies of apology lemmas in our dataset and shows that *sorry* is by far the most common apology lemma, which makes sense since *sorry* appears in example apologies in 11 of the 12 linguistics papers we reviewed for apology lemmas. After *sorry*, the most frequent apology lemmas are *mistake*, *fault*, *apology*, and *afraid*.

4.2.3.3 Developers' Self-Admitted Human Errors

RQ 6: Which human errors from literature do developers admit to? Which human errors from developer apologies do not exist in literature?

Developers apologized for 15 categories of human error that were not previously documented in research related to human errors in software engineering. Developers primarily apologize for mistakes, which is inconsistent with findings from previous literature.

We followed the categorization process outlined in Section 4.2.2.3 with a random sample of comments containing at least one apology lemma. We reviewed 332 apology comments resulting in 200 categorized human errors. The other 132 comments we reviewed could not be categorized, either because the comment author was a bot, the apology lemma(s) present were a false positive, the developer was not apologizing for a development mistake (*e.g.* apologizing for making a joke), the comment was deleted, the authoring user was deleted, or there simply was not enough context for us to accurately categorize the human error experienced.

While categorizing human errors, we identified 11 new categories of human error in the first 50 human error categorizations, three new human error categories in the second 50, and only one new category in the third 50. No new categories were created during the fourth set of 50.

We identified 15 human error categories that were not documented in previous research related to human errors in software engineering. These categories are described in Table 4.16. A comparison of software engineers' human errors from literature, from apologies on GitHub, and their overlap is included in Figure 4.11. We found that developers primarily admit to mistakes (66%), followed by slips (23%) and lapses (11%). As outlined in Table 4.15, the most frequent among developers' self-admitted human errors are `Time Management Errors (AM1)`, `Overlooking Documented Information (LS2)`, `Code Logic Errors (LM1)`, `External Communication Errors (LM5)`, `Working with Outdated Source Code (AL1)`, `Wrong Assumption Errors (LM3)`, `Workflow Order Errors (AM7)`, and `Internal Communication Errors (LM4)`. For two self-admitted slips, we did not have enough context to determine what kind of slip occurred.

The finding that developers primarily apologize for mistakes differs from previous findings that 49% of software engineers' human errors are slips, 17% are lapses, and 34% are mistakes [15]. Additionally, three categories of human error from T.H.E.S.E.—`Forgetting to Fix a Defect (LL1)`, `Forgetting to Remove Development Artifacts (LL2)`, and `Incomplete Domain Knowledge (LM2)`—were not applicable to the developer apologies we categorized, further suggesting that developers experience more mistakes than previously thought.

Our findings are also different from James Reason's finding that only 39% of human errors are mistakes [298]. We believe this difference is due to the highly plan-oriented nature of software engineering. While implementation is a key component of software engineering, the discipline as a whole depends heavily on effective planning and sticking

"I would not disable the test if flip is not present, but rather replace old line 167 by plot(m(end:-1:1)-0.5,'x–');. Sorry for not noticing that flip is not widely available. This works in octave (just checked in ancient 3.2.4) [284]."

Figure 4.10: Example Developer Apology from GitHub

This developer is apologizing for being unaware that a library feature is not standard.

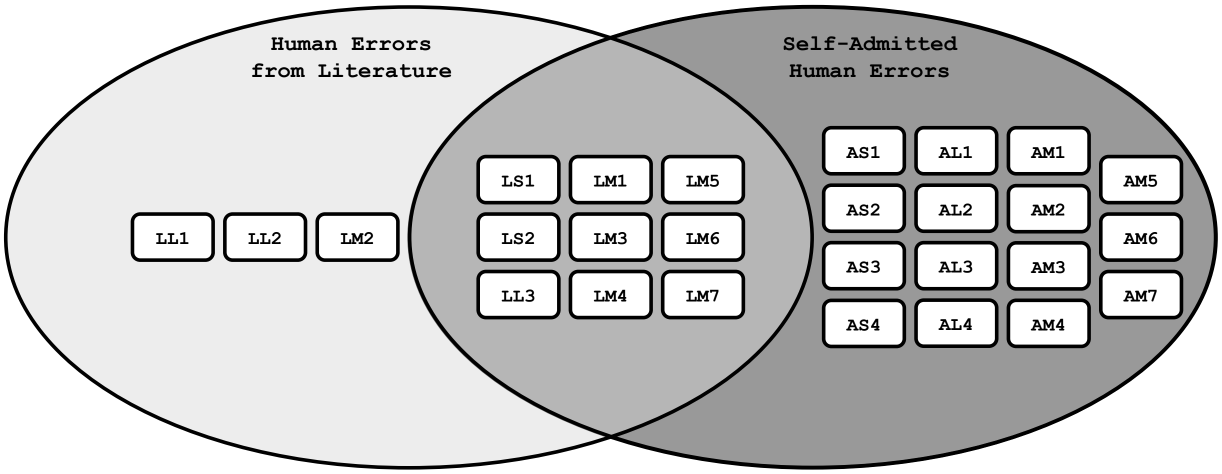


Figure 4.11: Venn Diagram of Human Errors from Literature and Developers' Apologies

Left: literature. Right: developers' apologies. See Table 4.6 and Table 4.16 for Human Error Identifiers.

to plans. Since mistakes are planning failures, it would make sense that a discipline heavily oriented around planning experiences more planning failures than attentional (*i.e.* slips) or memory (*i.e.* lapses) failures.

4.2.4 Limitations

When working with natural language, one concern is whether the data represents real human discourse. We chose to collect developers' conversations from GitHub, since pull requests (*i.e.* modern code review [113, 338]) and issue tracking are representative of open source software engineering activities. We also collected a large dataset (88.6 million developer comments) to ensure that our sample size for developer comments (especially apology comments) was ample. Additionally, we targeted conversations related to commonly used programming languages to ensure that our dataset contains typical developer conversations. While there are comments made by bots in our dataset, we took care when manually annotating apologies and categorizing human errors to identify and disregard bot comments. Further methodological details on how we collected our data can be found in Section 4.2.2.1.

We assessed our automatic apology classification in two ways. First, two researchers manually annotated a subset of apology comments using a well-defined and shared definition for apologies (see Section 4.2.2.2). Our very high inter-annotator agreement (Cohen's $\kappa = 0.94$) reflects this. Second, we evaluated our apology classifier using the well-known Switchboard corpus, which has apology annotations. In both cases, we examined false positives and false negatives, and had clear directions for improving our apology classification model. Even adjusting for our model's high false positive rate, our reported results remain valid: Of the 332 apology comments reviewed during human error categorization, 11 (3%) were manually identified as bots, and ten of those 11 bot comments were manually identified as apologies. If this percentage holds true throughout the entire dataset, then 77,426 out of 2.7 million apology comments were posted by bots, changing the percentage of apologies in our dataset to 2.82% (down from 2.91%), which is still over one thousand times greater than the 0.0004% of apologies found in the Switchboard corpus.

A high quality taxonomy should be unambiguous, complete, and sound/logical [192]. To reduce ambiguity, we defined a clear and rigid process for human error categorization in Section 4.2.2.2. Our categorization process handles special cases, such as using comment metadata to identify comments made by bots and users, and identifying false positive apology lemmas in quoted text, URLs, and email signatures. To address completeness, we reached a reasonable level of convergence by categorizing human errors for sets of 50 developer apologies until we were able to categorize 50 self-admitted human errors without creating a new category (as discussed in Section 4.2.3.3). Finally, our human error categories are both inspired by James Reason's well-established and well-defined Generic Error-Modelling System (GEMS) framework, and informed by previous findings in literature.

Table 4.15: Frequency of Self-Admitted Human Errors from Manual Categorization

Slips		**Lapses**		**Mistakes**			
ID	**Frequency**	**ID**	**Frequency**	**ID**	**Frequency**	**ID**	**Frequency**
LS2	22	**AL1**	15	**AM1**	29	**AM5**	7
AS1	9	**LL3**	3	**LM1**	21	**LM7**	5
AS3	7	**AL3**	2	**LM5**	17	**AM2**	3
AS4	3	**AL2**	1	**LM3**	14	**AM6**	3
AS2	2	**AL4**	1	**AM7**	12	**AM4**	2
Slip	2	**LL1**	0	**LM4**	10	**LM6**	1
LS1	1	**LL2**	0	**AM3**	8	**LM2**	0

We note that this study is based on open source software development discussions, and we do not differentiate between paid and unpaid developers; the results presented in Section 4.2.3 may not generalize to closed source software engineering. Getting access to closed source software development artifacts would have been challenging, and the nature of workplace environments and social constraints may not encourage self-admission of errors.

We must point out that plenty of self-admitted human errors exist beyond what is recorded in apologies, and plenty of human errors are not self-admitted. This is a matter of our intended scope, not a limitation of this study *per se*. We hope future researchers understand this when expanding upon this study.

4.2.5 Summary

In this section, we collected 88.6 million developer comments from commits, issues, and pull requests on GitHub. Using apology lemmas identified from linguistics literature, we created and tested an automated apology classification process. Finally, we categorized developers' self-admitted human errors using categories from Version 1 of T.H.E.S.E. Our key findings are:

- Apology lemmas can reliably identify developers' apologies in software engineering artifacts, with near perfect recall (99%) and high F1 score (87%)
- Developers apologize frequently, with an apology lemma density over one thousand times greater than speakers in the Switchboard corpus
- Developers' apologies are, on average, 36 words (67%) longer than their other comments
- Developers primarily apologize for mistakes (as opposed to slips and lapses), which differs from previous findings by Anu *et al.* [15] and Reason [298]
- We identified 15 categories of human error in software engineering not previously documented in literature

This study is an important step in examining developers' apologies and self-admitted human errors in software engineering artifacts. Our findings have revealed an incomplete understanding of software engineers' human errors by showing that human errors documented in controlled experiments (*i.e.* literature) are typically not representative of developers' self-admitted human errors.

Table 4.16: Human Errors in Software Engineering Identified from Developers' Apologies

ID	Category/Definition
Slips	
AS1	**Typos & Misspellings:** Typos and misspellings may occur in code comments, or when typing the name of a variable, function, or class.
AS2	**Multitasking Errors:** Errors resulting from multitasking.
AS3	**Hardware Interaction Errors:** Attention failures while using computer peripherals, such as mice, keyboard, and cables. Examples include copy/paste errors, clicking the wrong button, using the wrong keyboard shortcut, and incorrectly plugging in cables.
AS4	**Overlooking Proposed Code Changes:** Errors resulting from lack of attention during formal/informal code review.
Lapses	
AL1	**Working With Outdated Source Code:** Forgetting to git-pull (or equivalent in other version control systems), or using an outdated version of a library.
AL2	**Forgetting an Import Statement:** Forgetting to import a necessary library, class, variable, or function, or forgetting to include arguments in a function call.
AL3	**Forgetting Previous Development Discussion:** Errors resulting from forgetting details from previous development discussions.
AL4	**Forgetting to Implement a Feature:** Forgetting to implement a required feature.
Mistakes	
AM1	**Time Management Errors:** Errors resulting from a lack of time management, such as failing to allocate enough time for the implementation of a feature.
AM2	**Inadequate Testing:** Failure to implement necessary test cases, failure to consider necessary test inputs, or failure to implement a certain type of testing (*e.g.* unit, penetration, integration) when it is necessary.
AM3	**Incorrect/Insufficient Configuration:** Errors in configuration of libraries/frameworks/environments or errors related to missing configuration options.
AM4	**Internationalization/String Encoding Errors:** Errors related to internationalization and/or string/character encoding.
AM5	**Inadequate Experience Errors:** Errors resulting from inadequate experience with a language, library, framework, or tool.
AM6	**Insufficient Tooling Access Errors:** Errors resulting from not having sufficient access to necessary tooling. Examples include not having access to a specific operating system, library, framework, hardware device, or not having the necessary permissions to complete a development task.
AM7	**Workflow Order Errors:** Errors resulting from working out of order, such as implementing dependent features in the wrong order, implementing code before the design is stabilized, releasing code that is not ready to be released, or skipping a workflow step.

Chapter 5

Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.)

We begin this chapter by summarizing how T.H.E.S.E. was created. Then, we discuss an evaluation of T.H.E.S.E. and human error reflection based on human perceptions in a user study (Section 5.2), and a process for refining T.H.E.S.E. category definitions using semantic similarity (Section 5.3).

5.1 Creation

As we discussed in Chapter 4, Version 1 of T.H.E.S.E. was created by systematically identifying research related to human errors in software engineering, and aggregating those human errors into categories of slips, lapses, and mistakes. Version 1 of T.H.E.S.E. (shown in Table 4.6) consists of 12 categories of human error (2 slips, 3 lapses, and 7 mistakes). Next, we examined software engineers' self-admitted human errors in GitHub comments, and identified 15 new categories of human error (4 slips, 4 lapses, and 7 mistakes) that were not previously documented in literature (see Table 4.16). To begin this chapter, we present Version 2 of T.H.E.S.E. in Table 5.1, which is a combination of those 27 human errors, with the addition of three general *catch-all* categories (one each for slips, lapses, and mistakes).

5.2 Evaluation of T.H.E.S.E. by Software Engineering Students

5.2.1 Motivation & Research Questions

Software engineers must balance complex development activities spanning multiple engineering phases (*e.g.* requirements elicitation, design, implementation, testing, deployment, and maintenance) while working under strict constraints. Despite their best efforts, software engineers experience **human errors**. Human error theory from psychology has been studied in the context of software engineering, but human error assessment has yet to be adopted as part of typical post-mortem activities in software engineering.

Anu *et al.* [16] created and evaluated a Human Error Taxonomy (HET) aimed at requirements engineering. Graduate software engineering students inspected software requirements documents for faults. Students found, on average, 233% more faults [16] upon re-examining the requirements documents after receiving human error training. In a second study, students were able to identify more faults in requirements documents using HET than with traditional fault-checklists [19]. These are promising results, but the HET only addresses human errors during the requirements phase.

T.H.E.S.E. aggregates 30 categories (shown in Table 5.1) of human error spanning slips, lapses, and mistakes (failures of attention, memory, and planning, respectively) from human error literature and software engineering artifacts on GitHub. The broad goal of T.H.E.S.E. is to help software engineers confront and reflect on their human errors during all phases of software engineering, but the effectiveness of T.H.E.S.E. as a learning tool has yet to be assessed.

Our goal in this study was to evaluate T.H.E.S.E. as a learning tool for software engineering students. To that end, we conducted a user study involving five undergraduate software engineering students at the Rochester Institute

Table 5.1: Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.) Version 2

ID	Source	Category
Slips		
S01	Artifacts (Section 4.2)	**Typos & Misspellings**
S02	Literature (Section 4.1)	**Syntax Errors**
S03	Literature (Section 4.1)	**Overlooking Documented Information**
S04	Artifacts (Section 4.2)	**Multitasking Errors**
S05	Artifacts (Section 4.2)	**Hardware Interaction Errors**
S06	Artifacts (Section 4.2)	**Overlooking Proposed Code Changes**
S07	—	**General Attentional Failure**
Lapses		
L01	Artifacts (Section 4.2)	**Forgetting to Implement a Feature**
L02	Literature (Section 4.1)	**Forgetting to Fix a Defect**
L03	Literature (Section 4.1)	**Forgetting to Remove Development Artifacts**
L04	Artifacts (Section 4.2)	**Working with Outdated Source Code**
L05	Artifacts (Section 4.2)	**Forgetting an Import Statement**
L06	Literature (Section 4.1)	**Forgetting to Save Work**
L07	Artifacts (Section 4.2)	**Forgetting Previous Development Discussion**
L08	—	**General Memory Failure**
Mistakes		
M01	Literature (Section 4.1)	**Code Logic Errors**
M02	Literature (Section 4.1)	**Incomplete Domain Knowledge**
M03	Literature (Section 4.1)	**Wrong Assumption Errors**
M04	Literature (Section 4.1)	**Internal Communication Errors**
M05	Literature (Section 4.1)	**External Communication Errors**
M06	Literature (Section 4.1)	**Solution Choice Errors**
M07	Artifacts (Section 4.2)	**Time Management Errors**
M08	Artifacts (Section 4.2)	**Inadequate Testing**
M09	Artifacts (Section 4.2)	**Incorrect/Insufficient Configuration**
M10	Literature (Section 4.1)	**Code Complexity Errors**
M11	Artifacts (Section 4.2)	**Internationalization/String Encoding Errors**
M12	Artifacts (Section 4.2)	**Inadequate Experience Errors**
M13	Artifacts (Section 4.2)	**Insufficient Tooling Access Errors**
M14	Artifacts (Section 4.2)	**Workflow Order Errors**
M15	—	**General Planning Failure**

NOTE: *See Table 4.6 and Table 4.16 for category definitions.*

of Technology (RIT). In two experimental phases spanning 17 weeks total, participants self-reported and categorized (according to T.H.E.S.E.) human errors that they experienced during software development. We conducted weekly interviews and collected survey responses to evaluate T.H.E.S.E. We considered the following research questions:

RQ 7: **Ease of Use**
How clear, unambiguous, and simple is T.H.E.S.E. for software engineers to use?

RQ 8: **Comprehensiveness**
How well does T.H.E.S.E. cover human errors in software engineering?

RQ 9: **Assessment Value**
How well does T.H.E.S.E. facilitate human error reflection?

5.2.2 User Study Methodology

Our user study was conducted in two experimental phases. Phase 1 spanned four weeks from April 7, 2022 to May 5, 2022; Phase 2 spanned 13 weeks from September 9, 2022 to December 9, 2022.

5.2.2.1 Participants & Training

Participants in this study were undergraduate software engineering students at RIT. Demographic information was not collected. Participants were not aware of whom else was participating in the study. Participants will be referred to as Participant 1-5 throughout this work. Participants 1-2 participated in Phase 1; Participants 2-5[1] participated in Phase 2. Participants in Phase 1 were student employees of Dr. Meneely (this dissertation author's advisor) and thus were compensated at their normal hourly rate. Participants in Phase 2 were compensated with a $150 Amazon gift card[2] upon completion of the phase.

Each participant received one hour of human error training facilitated by the author of this dissertation. During training, the facilitator introduced participants to the concept of human error, emphasizing that everyone experiences human error [320] and that human errors are valuable learning opportunities [34, 199]. Training continued with (1) an introduction to slips, lapses, and mistakes, and (2) an introduction to T.H.E.S.E. and its categories. Participants were given example human errors with categories already labeled, and some practice human errors without labels. Participants worked through the steps to categorize human errors according to T.H.E.S.E. with the training facilitator and were given ample opportunity to ask questions.

5.2.2.2 Weekly Interviews

Following training, participants were tasked with documenting and reflecting on their human errors each week, leading up to a weekly interview. Throughout each week, participants documented the human errors that they experienced using a digital human error reporting form. In addition to the questions shown in Figure 5.1, the human error reporting form also displayed human error definitions and T.H.E.S.E. category definitions. During the weekly interviews, the study facilitator went through the participants' documented human errors to ask the follow-up questions shown in Figure 5.2. Weekly interviews were recorded for note-taking purposes.

5.2.2.3 Final Survey

After the final weekly interview, participants were given a final survey, in which all responses were anonymous. The final survey questions are shown in Figure 5.3, with questions (1), (2), (3), and (9) being answered on a Likert scale (strongly agree, somewhat agree, neither agree nor disagree, somewhat disagree, strongly disagree). Note that Participant 2 took the Phase 2 version of the final survey; they did not take the final survey at the end of Phase 1.

5.2.2.4 Phase 2 Improvements

After conducting Phase 1, we made improvements based on feedback from participants and our observations.

- Participants in Phase 1 found it difficult to answer Question 6 in Figure 5.2; for Phase 2, participants were able to see all of their previous submissions in a spreadsheet while answering this question.

[1]Participant 2 participated in both phases

[2]Participant 4 only participated in the first half of Phase 2 due to academic obligations, and thus was compensated with a $75 Amazon gift card.

1. Please briefly describe the human error that you experienced.
2. If the human error you experienced resulted in a defect that was committed, please provide a link (or Git commit hash) to the commit below.
3. Is your human error a slip, lapse, or mistake?
4. Now, please examine the Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.) and choose the specific human error that most accurately describes the human error you experienced. If you experienced multiple human errors, please submit this form once for each human error.
5. If there are other categories of human error that also describe the human error that you experienced, please note them here.
6. If you chose a "General" or "Other" category in Question 4, this question is required. Do you believe there is a missing human error category that better describes the human error that you experienced? If yes, please describe it below.
7. On a scale of 1 (not at all confident) to 5 (completely confident), how confident are you in your classification in the previous question?
8. Do you have any additional comments about this human error?

Figure 5.1: Human Error Reporting Form

Participants submitted a Google Form with these questions once per human error experienced. Human error type definitions and category descriptions were provided directly in the Google Form for quick reference. Questions 1, 2, 5, 6, and 8 were open ended. Questions 3 and 4 were multiple choice. Question 7 was answered on a 5-point Likert scale from not-at-all-confident to completely-confident.

Per Human Error:

1. Can you provide a little more detail about this human error?
2. Can you walk me through your process for categorization?
3. Was this human error particularly time consuming?
4. What might the consequences have been if this human error was not discovered?
5. How could this human error be avoided in the future?

In General:

6. How were the human errors you experienced this week different from or similar to previous weeks?
7. Did you experience any human errors that you could not categorize using T.H.E.S.E.? How would you categorize them?
8. Did you find yourself using human error terminology this week?
9. **Phase 1:** Did you have any difficulties that were not related to human error?
   Phase 2: Did you find yourself thinking in terms of human error as you experienced them, or was that a more reflective process?
10. Do you have any comments, questions, or feedback for me?

Figure 5.2: Weekly Interview Questions

Each week, participants were asked questions 1-5 for each human error that they reported, and then asked questions 6-10 in a general context. Participants in Phase 1 found Question 9 confusing, so this question was replaced for Phase 2. All questions were open ended.

1. This research project...
   (a) was engaging.
   (b) had clear instructions.
   (c) enhanced my understanding of my own human errors.
   (d) involved a reasonable time commitment.
   (e) was a valuable learning experience.
   (f) enhanced my understanding of human errors in SE.
   (g) reinforced theoretical concepts related to human error.
   (h) involved a useful human error reporting form.
   (i) would benefit other SE students.
   (j) involved meaningful weekly discussions.
2. The taxonomy...
   (a) had clear descriptions and examples.
   (b) was simple to use for classifying my human errors.
   (c) was general enough to apply to all SE phases.
   (d) led to meaningful reflection on my human errors.
   (e) made it easy to organize and confront my human errors.
   (f) would be a beneficial tool for professional software engineers.
   (g) had categories that adequately described my human errors.
   (h) led to unambiguous classifications.
   (i) was confusing.
   (j) was overwhelming.
3. The taxonomy adequately covers potential human errors during...
   (a) software requirements engineering.
   (b) software design.
   (c) software implementation.
   (d) software testing.
   (e) software deployment.
   (f) software maintenance.
4. Please elaborate on your answers in the previous question.
5. Which type of human error (*i.e.* slips, lapses, or mistakes) do you believe has the most impact (*i.e.* consequence) on a software project and why?
6. What strategies, activities, processes, or tools would you recommend for identifying and/or preventing **slips**? Which strategy, activity, process, or tool would you say can **best** identify and/or prevent most **slips**? Please be specific.
7. What strategies, activities, processes, or tools would you recommend for identifying and/or preventing **lapses**? Which strategy, activity, process, or tool would you say can **best** identify and/or prevent most **lapses**? Please be specific.
8. What strategies, activities, processes, or tools would you recommend for identifying and/or preventing **mistakes**? Which strategy, activity, process, or tool would you say can **best** identify and/or prevent most **mistakes**? Please be specific.
9. Based on your experience during this research project, please indicate your level of agreement with the following statements.
   (a) Human errors in SE typically fall into one category.
   (b) Human errors in SE often span multiple categories.
   (c) One human error in SE can lead to others.
   (d) I can usually identify **slips** as they occur.
   (e) I can usually identify **lapses** as they occur.
   (f) I can usually identify **mistakes** as they occur.
10. If multiple developers experienced the same human error, do you believe they would place it into the same category?
11. Please explain your answer to the previous question.
12. Do you believe that assessing human errors is a beneficial activity for software engineers?
13. Please explain your answer to the previous question.
14. Please explain what you learned about your own human errors during this project.
15. Please describe whether or not you feel like you can better avoid your own human errors in the future.
16. Please describe any difficulties you had while learning about human errors (slips, lapses, and mistakes) or while using T.H.E.S.E. to classify your human errors.
17. Do you have any recommendations for improving T.H.E.S.E.?
18. If this research project were repeated (in a class, for example), what changes would you like to see?
19. Would you be interested in seeing your human errors in some aggregate form? Please describe what form would be useful for you.
20. Do you have any additional comments, concerns, or suggestions that you would like to share?

Figure 5.3: Final Survey Questions

At the end of the user study, participants completed this survey. Questions 3-9 were added for Phase 2. Questions 1, 2, 3, and 9 were answered on a five-point Likert scale (strongly agree, somewhat agree, neither agree nor disagree, somewhat disagree, strongly disagree). Questions 10 and 12 were multiple choice (Yes / No / Maybe). All other questions were open ended.

Table 5.2: Summary of Human Errors Reported by User Study Participants

	Phase 1		**Phase 2**						**Phase 1**		**Phase 2**				
	P1	**P2**	**P2**	**P3**	**P4**	**P5**	**Total**		**P1**	**P2**	**P2**	**P3**	**P4**	**P5**	**Total**
Slips								***Mistakes***							
S01	—	—	—	1	1	14	16	**M01**	1	3	2	—	1	13	20
S02	—	1	—	4	—	—	5	**M02**	—	—	—	1	—	1	2
S03	1	2	—	2	—	4	9	**M03**	—	4	3	1	—	10	18
S04	—	1	—	—	—	—	1	**M04**	1	—	—	3	—	1	5
S05	—	1	—	—	—	10	11	**M05**	—	—	—	1	—	1	2
S06	—	—	—	—	—	—	0	**M06**	2	1	1	—	—	—	4
S07	1	1	1	—	1	3	7	**M07**	1	—	—	3	—	5	9
NEW	1	—	—	—	—	1	2	**M08**	—	—	—	—	—	—	0
Lapses								**M09**	—	1	—	1	2	2	6
L01	—	2	—	—	—	1	3	**M10**	3	—	—	—	—	—	3
L02	1	—	—	—	—	1	2	**M11**	—	—	—	—	—	1	1
L03	4	1	—	—	—	1	6	**M12**	—	—	—	2	2	6	10
L04	1	—	—	1	—	—	2	**M13**	—	—	—	2	—	1	3
L05	—	1	1	—	—	—	2	**M14**	—	—	—	1	—	2	3
L06	1	—	—	—	—	2	3	**M15**	1	—	—	—	—	—	1
L07	—	—	—	—	—	1	1	***Other***							
L08	—	—	—	1	—	4	5		—	—	—	—	—	—	0

- Participants found Question 9 in Figure 5.2 confusing, so we opted for a different question in Phase 2: *Did you find yourself thinking in terms of human error as you experienced them, or was that a more reflective process?*
- For Phase 2 we implemented a special portion for participants' final weekly interview. After completing the normal weekly interview process, participants were shown a spreadsheet of all of the human error categories from T.H.E.S.E. that they *did not* document human errors for, and were asked (1) whether they have experienced that human error at any point during their software engineering career, and (2) if not, if they believe it is possible for software engineers to experience that human error.
- Questions 3-9 in Figure 5.3 were added for Phase 2 to collect additional information.
- Question 8 in Figure 5.1 was added for Phase 2.

5.2.2.5 Institutional Review Board Approval

Institutional Review Board approval for this research involving human subjects was granted by the Human Subjects Research Office at the Rochester Institute of Technology on March 18, 2022. The full form is provided in Appendix B. Participants read and signed an informed consent (see Appendix C) form acknowledging that (1) their participation was entirely voluntary and had no impact on their grades, and (2) their survey responses would be published in an anonymized format. All data released from our user study has been anonymized by replacing any personally identifiable information with participant identifiers. Video recordings taken during weekly interviews will never be released.

5.2.3 Results

Results for Likert and multiple choice questions in the final survey (Fig 5.3) are shown in Table 5.3. Responses to open-ended questions are included in Appendix E. Participants reported a total of 162 human errors (38 during Phase 1; 124 during Phase 2). Categories of human error reported are shown in Table 5.2, including a new category (discussed in Section 5.2.3.1). While discussing results, "agree" and "disagree" should be read as "somewhat or strongly agree" and "somewhat or strongly disagree", respectively.

5.2.3.1 Ease of Use

RQ 7: How clear, unambiguous, and simple is T.H.E.S.E. for software engineers to use?

Participants indicated that T.H.E.S.E. has clear definitions/examples and makes it easy to organize and confront their human errors.

All participants agreed that T.H.E.S.E. (1) has clear definitions and examples, (2) is simple to use for classifying human errors, and (3) makes it easy to organize and confront their human errors. All participants disagreed with the notion of T.H.E.S.E. being confusing or overwhelming. When asked if T.H.E.S.E. led to unambiguous classifications, two participants agreed, while three neither agreed nor disagreed. One participant anonymously indicated that developers' "perception of the situation might lead to different placements" of human errors.

The results indicate that T.H.E.S.E. is generally easy to use, but there may be some ambiguity in the human error reflection process due to developers' perceptions and experiences.

5.2.3.2 Comprehensiveness

RQ 8: How well does T.H.E.S.E. cover human errors in software engineering?

Participants indicated that T.H.E.S.E. covers all phases of software engineering, but experienced one category of human error not captured in T.H.E.S.E.

All participants agreed that T.H.E.S.E. has categories that adequately described their human errors. 4/5 participants agreed (one neither agreeing nor disagreeing) that T.H.E.S.E. is general enough to apply to all software engineering activities, but that T.H.E.S.E. does not necessarily adequately cover the software design and maintenance phases.

During Phase 1, one of the 38 reported human errors could not be placed into an existing category:

> **Participant 1:** *"Using a hard-coded value for a width property when one was already set. The width property... was like a calculated value, like based on the actual size of the page... so that calculated value was like 365 and that would have been better for the page, and then a few places I just forgot about it totally and I said 350 and it was just like hardcoded in there."*

As a result, we created a new slip category: `Overlooking Existing Functionality:` *Errors resulting from overlooking existing functionality, such as reimplementing variables, functions, and classes that already exist, or reimplementing functionality that already exists in a standard library.*

`Overlooking Proposed Code Changes (S06)` and `Inadequate Testing (M08)` were not experienced during the user study, but participants indicated that these human errors could be experienced by other software developers. The results indicate that T.H.E.S.E. is mostly comprehensive, but there may be some human error edge cases that are not covered.

5.2.3.3 Assessment Value

RQ 9: How well does T.H.E.S.E. facilitate human error reflection?

Participants indicated that human error assessment assisted by T.H.E.S.E. is a beneficial software engineering activity that leads to meaningful human error reflection.

All participants agreed that (1) T.H.E.S.E. leads to meaningful reflection of their human errors and (2) human error assessment with T.H.E.S.E. enhanced their understanding of human errors in software engineering and of their own human errors. 4/5 participants agreed (one neither agreeing nor disagreeing) that (1) T.H.E.S.E. would be a beneficial tool for professional software engineers, (2) human error assessment would benefit other software engineering students, (3) human error assessment with T.H.E.S.E. is a valuable learning experience, and (4) the weekly discussions were meaningful. 3/5 participants believe that assessing human errors in software engineering is beneficial, with the other 2/5 indicating that human error assessment could be costly in terms of time.

These results indicate that human error assessment assisted by T.H.E.S.E. is a beneficial learning experience for software engineering students and may also provide benefit to professional software developers.

Table 5.3: Final Survey Responses for Likert and Yes/No/Maybe Questions

	Survey Respondent				
Q#	**A**	**B**	**C**	**D**	**E**
(1-a)	Somewhat Agree	Somewhat Agree	Somewhat Agree	Strongly Agree	Strongly Agree
(1-b)	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree
(1-c)	Strongly Agree	Somewhat Agree	Strongly Agree	Strongly Agree	Strongly Agree
(1-d)	Strongly Agree	Somewhat Agree	Somewhat Agree	Strongly Agree	Strongly Agree
(1-e)	Neither	Strongly Agree	Somewhat Agree	Strongly Agree	Somewhat Agree
(1-f)	Strongly Agree	Somewhat Agree	Strongly Agree	Strongly Agree	Strongly Agree
(1-g)	Strongly Agree	Neither	Strongly Agree	Strongly Agree	Somewhat Agree
(1-h)	Strongly Agree	Somewhat Agree	Strongly Agree	Strongly Agree	Strongly Agree
(1-i)	Somewhat Agree	Neither	Somewhat Agree	Somewhat Agree	Strongly Agree
(1-j)	Neither	Somewhat Agree	Somewhat Agree	Strongly Agree	Strongly Agree
(2-a)	Strongly Agree	Somewhat Agree	Strongly Agree	Strongly Agree	Strongly Agree
(2-b)	Strongly Agree	Somewhat Agree	Strongly Agree	Strongly Agree	Somewhat Agree
(2-c)	Strongly Agree	Neither	Somewhat Agree	Strongly Agree	Somewhat Agree
(2-d)	Strongly Agree	Somewhat Agree	Strongly Agree	Strongly Agree	Strongly Agree
(2-e)	Strongly Agree	Somewhat Agree	Strongly Agree	Strongly Agree	Strongly Agree
(2-f)	Strongly Agree	Somewhat Agree	Somewhat Agree	Neither	Strongly Agree
(2-g)	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Somewhat Agree
(2-h)	Somewhat Agree	Neither	Neither	Strongly Agree	Neither
(2-i)	Strongly Disagree	Somewhat Disagree	Somewhat Disagree	Strongly Disagree	Somewhat Disagree
(2-j)	Strongly Disagree	Somewhat Disagree	Somewhat Disagree	Strongly Disagree	Somewhat Disagree
(3-a)	—	Neither	Strongly Agree	Strongly Agree	Somewhat Agree
(3-b)	—	Somewhat Disagree	Strongly Agree	Somewhat Agree	Strongly Agree
(3-c)	—	Strongly Agree	Strongly Agree	Strongly Agree	Somewhat Agree
(3-d)	—	Strongly Agree	Strongly Agree	Somewhat Agree	Strongly Agree
(3-e)	—	Somewhat Agree	Strongly Agree	Strongly Agree	Strongly Agree
(3-f)	—	Somewhat Agree	Strongly Agree	Somewhat Disagree	Strongly Agree
(9-a)	—	Somewhat Disagree	Somewhat Disagree	Neither	Somewhat Disagree
(9-b)	—	Neither	Strongly Agree	Somewhat Agree	Somewhat Agree
(9-c)	—	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree
(9-d)	—	Somewhat Agree	Strongly Agree	Somewhat Disagree	Somewhat Agree
(9-e)	—	Neither	Strongly Agree	Somewhat Agree	Neither
(9-f)	—	Strongly Disagree	Strongly Agree	Strongly Agree	Somewhat Agree
(10)	Yes	Maybe	Yes	Yes	Maybe
(12)	Yes	Yes	Maybe	Maybe	Yes

5.2.4 Improvements to T.H.E.S.E.

In response to participant feedback, we updated T.H.E.S.E. to Version 3 (Table 5.4). These changes were made to make category definitions more clear, provide more examples, and broaden the scope of `Forgetting to Finish a Development Task (L03)`. Specifically, we made the following changes:

- **Identifier Changes:**
  - `General Attentional Failure (S07)` → `General Attentional Failure (S08)`
- **New Categories:**
  - `Overlooking Existing Functionality (S07)` – *Errors resulting from overlooking existing functionality, such as reimplementing variables, functions, and classes that already exist, or reimplementing functionality that already exists in a standard library.*
- **Name Changes:**
  - `Forgetting to Implement a Feature (L01)` → `Forgetting to Finish a Development Task (L01)`
- **Definition Updates:**
  - `Typos & Misspellings (S01)` – *Typos and misspellings may occur in code comments, documentation (and other development artifacts), or when typing the name of a variable, function, or class. Examples include misspelling a variable name, writing down the wrong number/name/word during requirements elicitation, referencing the wrong function in a code comment, and inconsistent whitespace (that does not result in a syntax error).*
  - `Syntax Errors (S02)` – *Any error in coding language syntax that impacts the executability of the code. Note that Logical Errors (e.g. += instead of +) are not Syntax Errors. Examples include mixing tabs and spaces (e.g. Python), unmatched brackets/braces/parenthesis/quotes, and missing semicolons (e.g. Java).*
  - `Overlooking Documented Information (S03)` – *Errors resulting from overlooking (internally and externally) documented information, such as project descriptions, stakeholder requirements, API/library/-tool/framework documentation, coding standards, programming language specifications, bug/issue reports, and looking at the wrong version of documentation or documentation for the wrong project/software.*
  - `Forgetting to Finish a Development Task (L01)` – *Forgetting to finish a development task. Examples include forgetting to implement a required feature, forgetting to finish a user story, and forgetting to deploy a security patch.*
  - `Forgetting to Remove Development Artifacts (L03)` – *Forgetting to remove debug log files, dead code, informal test code, commented out code, test databases, backdoors, etc. Examples include leaving unnecessary code in the comments, and leaving notes in internal development documentation.*
  - `Code Logic Errors (M01)` – *A code logic error is one in which the code executes (i.e. actually runs), but produces an incorrect output/behavior due to incorrect logic. Examples include using incorrect operators (e.g. += instead of +), erroneous if/else statements, incorrect variable initializations, problems with variable scope, and omission of necessary logic.*
  - `Incorrect/Insufficient Configuration (M09)` – *Errors in configuration of libraries/frameworks/environments or errors related to missing configuration options. Examples include misconfigured IDEs, improper directory structure for a specific programming language, and missing SSH keys.*

5.2.5 Limitations

The main limitations of this study are the small number of participants and their relative inexperience. To mitigate the smaller number of participants, we opted for a longer observation experiment so that we could capture a greater sampling of their process. For the experience issue, these students are part of a program that requires cooperative education, so they had some industry experience in the field of software engineering before this study. Finally, James Reason noted that "a taxonomy is usually made for a specific purpose, and no single scheme is likely to satisfy all needs [298]"—a taxonomy such as T.H.E.S.E. is likely never to be fully comprehensive, as humankind never stops being creative in its missteps.

5.2.6 Summary

In this work, our goal was to evaluate T.H.E.S.E. as a learning tool for software engineering students. Over 17 weeks, we conducted two experimental phases in which five software engineering students documented, categorized (according to T.H.E.S.E.), and reflected on their human errors. We conducted weekly interviews and collected survey responses from participants. All participants agree that (1) T.H.E.S.E. has clear definitions and makes it easy to organize and confront their human errors, (2) categories in T.H.E.S.E. adequately describe their human errors and are general enough to apply to all software engineering activities, and (3) human error assessment facilitated by T.H.E.S.E. was a valuable learning experience which led to meaningful human error reflection, and would benefit other software engineering students and professional software developers. The key contributions of this research are as follows:

- Evaluation of T.H.E.S.E. as a learning tool for software engineering students
- Identification of a new category of human error in software engineering not previously documented
- Anonymized human error reports and insights from software engineering students

5.3 Semi-Automated Refinement of T.H.E.S.E. Categories

5.3.1 Motivation & Research Questions

Results from our user study (Section 5.2.3) are promising, indicating that human error assessment with T.H.E.S.E. is a valuable learning experience for software engineers. However, one challenge for adoption of new processes/technologies is ease of use [74, 343][3]. While participants indicated that T.H.E.S.E. is easy to use, they also indicated that there may be some ambiguity in human error categorization using T.H.E.S.E., possibly due to software developers' personal perceptions and experiences.

To address potential ambiguity in T.H.E.S.E. category definitions, we used state-of-the-art Sentence-BERT models—models capable of representing sentence-level meaning in natural language—to determine where our human error data shared a semantic *dissimilarity* to T.H.E.S.E. category definitions. When such a dissimilarity was found, we examined the misclassified data and circled back to improve our category definitions. From a modeling standpoint, this resulted in a model that is overfit to the data. However, from a T.H.E.S.E. standpoint, we have improved the completeness of our definitions. For example (inspired by our results in Section 5.2.4), a general purpose Sentence-BERT model might not see a strong similarity between sentences containing the words "whitespace" and "typo" in similar contexts, so this methodology would lead us to add "whitespace" to our improved definition for `Typos & Misspellings (S01)`. The automated portion of this study is in finding dissimilarities, and the manual portion is in refining the category definitions, hence our "semi-automated" approach. Our aim was not to train a perfect classifier, but instead to use semantic similarity to guide improvement of T.H.E.S.E. category descriptions.

However, this experiment also provided us an opportunity to explore whether automated human error categorization is feasible, which could improve the likelihood of software engineers adopting human error reflection with T.H.E.S.E. and provide a valuable asset to software quality tools. Our research questions are as follows:

RQ 10: Category Ambiguity
How well can ambiguity of T.H.E.S.E. category descriptions be reduced based on semantic similarity?

RQ 11: Assisted Categorization
How useful is Sentence-BERT for refining human error definitions?

[3]Agarwal *et al.* [2] explored this notion in the context of software engineering, but did not find a statistically significant correlation between ease of use and software engineers' attitude toward using a new programming language. However, they noted that it is still important to consider ease of use for the adoption of new technologies.

Table 5.4: Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.) Version 3

ID	Source	Category
Slips		
S01	Artifacts (Section 4.2)	**Typos & Misspellings**
S02	Literature (Section 4.1)	**Syntax Errors**
S03	Literature (Section 4.1)	**Overlooking Documented Information**
S04	Artifacts (Section 4.2)	**Multitasking Errors**
S05	Artifacts (Section 4.2)	**Hardware Interaction Errors**
S06	Artifacts (Section 4.2)	**Overlooking Proposed Code Changes**
S07	User Study (Section 5.2)	**Overlooking Existing Functionality**
S08	—	**General Attentional Failure**
Lapses		
L01	Artifacts (Section 4.2)	**Forgetting to Finish a Development Task**
L02	Literature (Section 4.1)	**Forgetting to Fix a Defect**
L03	Literature (Section 4.1)	**Forgetting to Remove Development Artifacts**
L04	Artifacts (Section 4.2)	**Working with Outdated Source Code**
L05	Artifacts (Section 4.2)	**Forgetting an Import Statement**
L06	Literature (Section 4.1)	**Forgetting to Save Work**
L07	Artifacts (Section 4.2)	**Forgetting Previous Development Discussion**
L08	—	**General Memory Failure**
Mistakes		
M01	Literature (Section 4.1)	**Code Logic Errors**
M02	Literature (Section 4.1)	**Incomplete Domain Knowledge**
M03	Literature (Section 4.1)	**Wrong Assumption Errors**
M04	Literature (Section 4.1)	**Internal Communication Errors**
M05	Literature (Section 4.1)	**External Communication Errors**
M06	Literature (Section 4.1)	**Solution Choice Errors**
M07	Artifacts (Section 4.2)	**Time Management Errors**
M08	Artifacts (Section 4.2)	**Inadequate Testing**
M09	Artifacts (Section 4.2)	**Incorrect/Insufficient Configuration**
M10	Literature (Section 4.1)	**Code Complexity Errors**
M11	Artifacts (Section 4.2)	**Internationalization/String Encoding Errors**
M12	Artifacts (Section 4.2)	**Inadequate Experience Errors**
M13	Artifacts (Section 4.2)	**Insufficient Tooling Access Errors**
M14	Artifacts (Section 4.2)	**Workflow Order Errors**
M15	—	**General Planning Failure**

NOTE: *See Table 4.6, Table 4.16, and Section 5.2.4 for category definitions.*

Table 5.5: Summary of Human Error Descriptions Used for Classification

Category	Frequency	Percentage	Category	Frequency	Percentage
Slips	**96**	**26.087%**	***Mistakes***	**225**	**61.141%**
S01	25	6.793%	**M01**	41	11.141%
S02	6	1.630%	**M02**	2	0.543%
S03	32	8.696%	**M03**	32	8.696%
S04	2	0.543%	**M04**	16	4.348%
S05	18	4.891%	**M05**	19	5.163%
S06	3	0.815%	**M06**	7	1.902%
S07	2	0.543%	**M07**	39	10.598%
S08	9	2.446%	**M08**	3	0.815%
Lapses	**47**	**12.772%**	**M09**	14	3.804%
L01	4	1.087%	**M10**	8	2.174%
L02	2	0.543%	**M11**	3	0.815%
L03	6	1.630%	**M12**	17	4.620%
L04	17	4.620%	**M13**	6	1.630%
L05	3	0.815%	**M14**	16	4.348%
L06	6	1.630%	**M15**	1	0.272%
L07	4	1.087%	***Other***	**0**	**0.000%**
L08	5	1.359%			

5.3.2 Methodology

5.3.2.1 Data Selection

This study used two sources of data: (1) 200 self-admitted human errors from GitHub issues, pull-requests, and comments, and (2) 168 human error descriptions[4] from our user study (Section 5.2). These 368 human error descriptions are summarized in Table 5.5. These human error descriptions were manually categorized according to T.H.E.S.E. as described in Section 4.2.2.3 for the GitHub comments and Section 5.2.2.2 for the user study descriptions.

5.3.2.2 Model Selection

We computed Sentence-BERT embeddings for each human error type description—concatenated definitions for T.H.E.S.E. categories corresponding to that type (target class)—and for each human error description in our dataset (data point). We only consider human error types (*i.e.* slip, lapse, mistake) as target classes, since most human error categories (*e.g.* `S01`, `M01`, *etc.*) have fewer than 10 examples in our dataset.

Next, we computed the cosine similarity between the Sentence-BERT embeddings for each data point and each target class. The higher the cosine similarity between two Sentence-BERT embeddings, the closer their meaning. For example, the human error description *"I used just = in my conditional instead of ==. The IDE didn't catch it. It compiled, too."* has a cosine similarity of 0.4722[5] with the type description for *mistakes*.

We repeated this process for 21 pretrained Sentence-BERT models (shown in Table 5.8). We chose these models because they were either "extensively evaluated for their quality to embedded [sic] sentences [302]", had thousands of downloads from HuggingFace, or were fine-tuned using software engineering adjacent (*e.g.* security) natural language. The model that results in the highest F1 score for a target class was considered the best model for classifying that target class. In total, we conducted 8 classification experiments (see Table 5.7) for 21 pretrained Sentence-BERT models (168 different combinations). Experiments varied by three factors:

- **Data:** Experiments either used all 368 data points or only 168 human errors documented in our user study. We examined results without apology comments because they tend to be less detailed than the human errors documented in our user study.

[4]Participants only formally documented 162 human errors, but 6 additional human errors were informally reported by Participant 4 in weekly interviews.

[5]Using the SB08 model.

Table 5.6: Best Sentence-BERT Models for Classification Before and After Improving T.H.E.S.E. Definitions

	Slips		**Lapses**		**Mistakes**	
	Before	**After**	**Before**	**After**	**Before**	**After**
Model	SB03	SB03	SB12	SB20	SB09	SB13
Median Precision	0.368	0.418	0.216	0.182	0.662	0.593
Median Recall	0.500	0.526	0.564	0.809	0.631	0.511
Median F1	0.455	0.475	0.318	0.295	0.638	0.550
Frequency in Dataset	96/368		47/368		225/368	

Table 5.7: Summary of Classification Experiments

	Data Configurations		
Experiment	**Class**	**Data**	**Preprocessing**
1	Multiclass	User Study & Apologies	No
2	Multiclass	User Study & Apologies	Yes
3	Multiclass	User Study	No
4	Multiclass	User Study	Yes
5	Binary	User Study & Apologies	No
6	Binary	User Study & Apologies	Yes
7	Binary	User Study	No
8	Binary	User Study	Yes

- **Preprocessing:** Experiments either used raw human error descriptions or preprocessed descriptions with text lowercased, punctuation removed, and non-space whitespace characters (*e.g.* newlines) removed. Sentence-BERT has built in tokenization, and thus expects unprocessed data. We do not expect improvements from our preprocessing.

- **Classification Type:** Experiments were either multiclass (*i.e.* slip vs. lapse vs. mistake) or binary (*i.e.* slip vs. not-slip, lapse vs. not-lapse, mistake vs. not-mistake). For binary experiments, we considered the other two classes to be the same. For example, for the binary slip classifier, the data labels for lapses and mistakes were changed to *not-slip*, and we compute cosine similarities for two target classes—slip and not-slip—using the Sentence-BERT embedding for the slip description and the concatenated lapse-and-mistake—not-slip—description.

In practice, we would anticipate that high precision is generally best for a recommendation system that uses T.H.E.S.E., but the underrepresented lapse class in our dataset resulted in high recall for lapses for the models we evaluated. To avoid over-classifying human errors as lapses, we selected the models that are well-balanced at classifying slips, lapses, and mistakes, respectively, based on median F1 score. The models we selected are summarized in Table 5.6. The full precision, recall, and F1 scores for all 8 experiments are provided in Appendix F.

We must note that our test data (human error descriptions from our user study and GitHub apologies) is our training data; our results may not generalize to new data. However, this is acceptable for our use case, as our goal is not robust classification, but instead to create a tool to assist us in refining taxonomy definitions.

5.3.2.3 Classification Process

For **RQ11**, we used an ensemble of Sentence-BERT models from Table 5.6 for classification. Our ensemble classification process is visually summarized in Figure 5.4 and outlined below:

- **Step 1: Compute Embeddings**: Compute the Sentence-BERT embeddings for each human error type description, T.H.E.S.E. category definition, and new human error description to be classified.

- **Step 2: Compare Cosine Similarities for Types**: Compare cosine similarities between the new human error description and each human error type description.

Table 5.8: Pretrained Sentence-BERT Models

ID	Name	ID	Name
SB01	all-mpnet-base-v2 [88]	**SB12**	distiluse-base-multilingual-cased-v1 [263]
SB02	multi-qa-mpnet-base-dot-v1 [90]	**SB13**	distiluse-base-multilingual-cased-v2 [264]
SB03	all-distilroberta-v1 [92]	**SB14**	nikcheerla/nooks-amd-detection-v2-full [259]
SB04	all-MiniLM-L12-v2 [91]	**SB15**	jhgan/ko-sroberta-multitask [151]
SB05	multi-qa-distilbert-cos-v1 [89]	**SB16**	ceggian/sbert_pt_reddit_softmax_512 [59]
SB06	all-MiniLM-L6-v2 [96]	**SB17**	BlueAvenir/sti_security_class_model [170]
SB07	multi-qa-MiniLM-L6-cos-v1 [93]	**SB18**	jhgan/ko-sbert-sts [150]
SB08	paraphrase-multilingual-mpnet-base-v2 [265]	**SB19**	nikcheerla/nooks-amd-detection-realtime [258]
SB09	paraphrase-albert-small-v2 [95]	**SB20**	LaBSE [155]
SB10	paraphrase-multilingual-MiniLM-L12-v2 [266]	**SB21**	kwoncho/ko-sroberta-multitask-suspicious [175]
SB11	paraphrase-MiniLM-L3-v2 [94]		

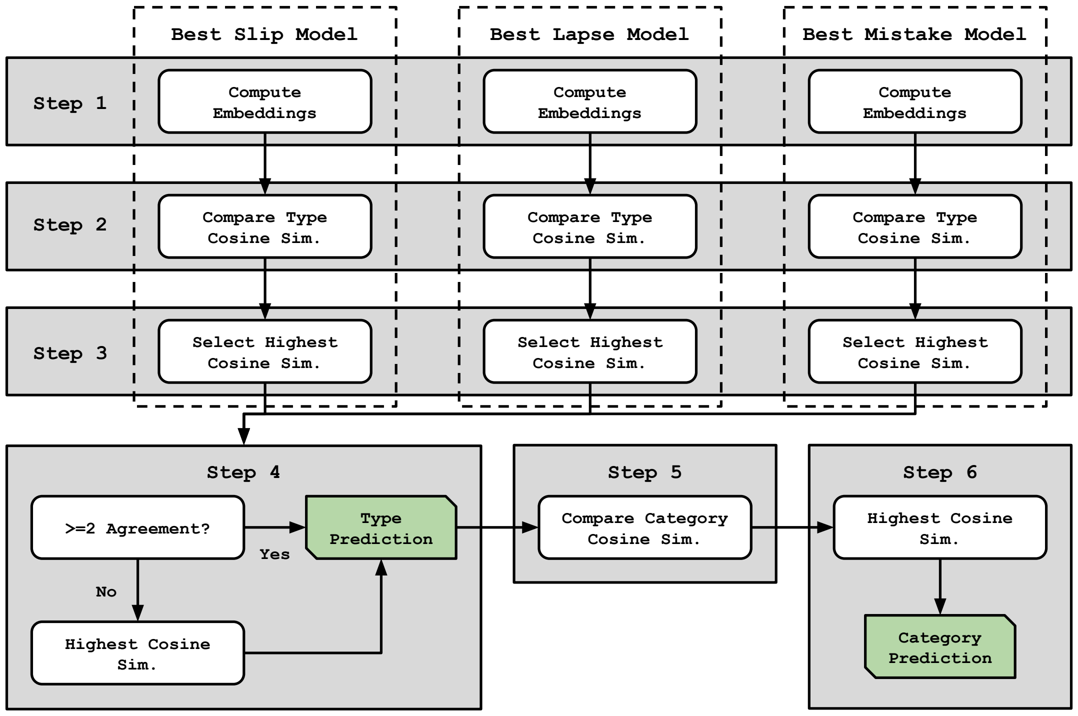


Figure 5.4: Sentence-BERT Based Ensemble Classifier for Human Errors
The full classification process is described in Section 5.3.2.3. Key: Sim.—Similarity.

Table 5.9: Ensemble Classification Results

Type	**FP**	**TP**	**FN**	**TN**	**Precision**	**Recall**	**F1**
Before Improving T.H.E.S.E. Definitions							
Slips	63	46	51	208	0.422	0.474	0.447
Lapses	136	31	16	185	0.186	0.660	0.290
Mistakes	28	64	160	116	0.696	0.286	0.405
Average	75.7	47.0	75.7	169.7	0.435	0.473	0.381
Improvement Attempt 1 – Extra Examples							
Slips	70	50	47	201	0.417	0.515	0.461
Lapses	132	28	19	189	0.175	0.596	0.271
Mistakes	26	62	162	118	0.705	0.277	0.397
Average	76.0	46.7	76.0	169.3	0.432	0.463	0.376
Improvement Attempt 2 – Extended Descriptions from CWE							
Slips	52	50	47	219	0.490	0.515	0.503
Lapses	144	28	19	177	0.163	0.596	0.256
Mistakes	25	69	155	119	0.734	0.308	0.434
Average	73.7	49.0	73.7	171.7	0.462	0.473	0.398
Improvement Attempt 3 – Extended Descriptions from Anu et al. [12]							
Slips	53	52	45	218	0.495	0.536	0.515
Lapses	142	28	19	179	0.165	0.596	0.258
Mistakes	23	70	154	121	0.753	0.312	0.442
Average	72.7	50.0	72.7	172.7	0.471	0.481	0.405

- **Step 3: Select Intermediate Type Classification**: The human error type with the highest cosine similarity score is selected. For example, if the cosine similarities are 0.2751, 0.1852, and 0.4722 for slip, lapse, and mistake, respectively, then the selected classification is *mistake*.

Steps 1-3 were repeated three times—once using the best slips model, once with the best lapses model, and once with the best mistakes model—resulting in three classifications.

- **Step 4: Select Type Classification**: Steps 1-3 result in three intermediate classifications. If two or more classifications match (*e.g.* slip, slip, mistake), then we decide that the new description is that class. Otherwise, if there is no agreement between classifications, we take the classification with the highest overall cosine similarity.
- **Step 5: Compare Cosine Similarities for Categories**: Repeat Step 2 using the T.H.E.S.E. category definitions associated with the type classification from Step 4. For example, if the slip type was selected in Step 4, compare cosine similarities between the new human error description and the `S01-S08` category definitions.
- **Step 6: Select Category Classification**: Select the category associated with the highest cosine similarity from Step 5.

5.3.2.4 Improving T.H.E.S.E. Definitions & Examples

We examined false-positive and false-negative classifications from the selected ensemble classifier for patterns and opportunities to improve T.H.E.S.E. definitions. A summary of classifications from our ensemble classifier is provided in Table 5.10. For our first attempt at improving T.H.E.S.E. category definitions, we examined misclassifications for seven categories—`S04`, `S07`, `L05`, `M02`, `M08`, `M11`, and `M15`—that were always incorrectly classified, improved their definitions by adding examples, and repeated classification with our ensemble classifier.

For our second attempt at improving T.H.E.S.E. category definitions, we took a different approach. Instead of modifying category definitions directly, we created an extended description for each category. To populate extended descriptions, we identified CWE entries that fall under categories of human error, and copied the CWE titles and

Table 5.10: Summary of Human Error Classifications Before Improving T.H.E.S.E. Definitions

		Assigned					Assigned		
	Total	**Slip**	**Lapse**	**Mistake**		**Total**	**Slip**	**Lapse**	**Mistake**
Slips					***Mistakes***				
S01	25	19	3	3	**M01**	41	13	9	19
S02	6	6	0	0	**M02**	2	0	2	0
S03	32	13	12	7	**M03**	32	12	10	10
S04	2	0	2	0	**M04**	16	2	12	2
S05	18	7	7	4	**M05**	19	4	8	7
S06	3	1	1	1	**M06**	7	1	2	4
S07	2	0	1	1	**M07**	39	5	27	7
S08	9	2	4	3	**M08**	3	2	1	0
Lapses					**M09**	14	2	9	3
L01	4	0	4	0	**M10**	8	2	3	3
L02	2	1	1	0	**M11**	3	3	0	0
L03	6	1	3	2	**M12**	17	5	6	6
L04	17	2	12	3	**M13**	6	1	4	1
L05	3	2	0	1	**M14**	16	1	13	2
L06	6	0	5	1	**M15**	1	1	0	0
L07	4	2	2	0	***Other***				
L08	5	1	4	0		—	—	—	—

descriptions into the relevant extended description. We repeated our ensemble classification using concatenated definitions and extended descriptions for each T.H.E.S.E. category.

For our third attempt at improving T.H.E.S.E. category definitions, we added human error category titles and descriptions from the HET [12] to the extended descriptions for `S08`, `L08`, and `M15` (general slips, lapses, and mistakes, respectively). We repeated our ensemble classification. Note that while we used category descriptions from HET to classify slips, lapses, and mistakes, we did not use them to classify T.H.E.S.E. categories to avoid over-classifying human errors as general slips, lapses, or mistakes.

Classification results from all three attempts are shown in Table 5.9. Extended descriptions for T.H.E.S.E. categories are provided in Appendix G. The improvements we made to T.H.E.S.E. category definitions are as follows:

- `Multitasking Errors (S04)` – *Errors resulting from multitasking, i.e. working on multiple software engineering tasks at the same time.*
- `Overlooking Proposed Code Changes (S06)` – *Attention failures while using computer peripherals, such as mice, keyboard, and cables. Examples include copy/paste errors, clicking the wrong button, using the wrong keyboard shortcut, and incorrectly plugging in cables.*
- `Overlooking Existing Functionality (S07)` – *Errors resulting from overlooking existing functionality, such as reimplementing or duplicating variables, functions, and classes that already exist, or reimplementing functionality that already exists in a standard library. Other examples include deleting necessary variables, functions, and classes.*
- `Forgetting an Import Statement (L05)` – *Forgetting to import a necessary library, class, variable, or function, or forgetting to access a property, attribute, or argument. Examples include forgetting to import python's sys library, forgetting to include a header file in C, or forgetting to pass an argument to a function.*
- `Incomplete Domain Knowledge (M02)` – *Errors resulting from incomplete knowledge of the software system's target domain (e.g. banking, astrophysics). Examples include planning/designing a system without understanding the nuances of the domain.*
- `Internal Communication Errors (M04)` – *Errors resulting from inadequate communication between development team members. Examples include misunderstanding development discussion, misinterpreting or providing*

ambiguous instructions, communicating using the wrong medium (e.g. oral vs. written), or communicating ineffectively (e.g. too formal/informal, too much unnecessarily complex language, hostile language/body language).

- `External Communication Errors (M05)` – *Errors resulting from inadequate communication with project stakeholders or third-party contractors. Examples include providing ambiguous or unclear directions to third-parties or users, or misinterpreting stakeholder feedback, communicating using the wrong medium (e.g. oral vs. written), or communicating ineffectively (e.g. too formal/informal, too much unnecessarily complex language, hostile language/body language).*
- `Solution Choice Errors (M06)` – *Misunderstood problem-solving methods/techniques result in analyzing the problem incorrectly and choosing the wrong solution. For example, choosing to implement a database system in Python rather than using SQL, or choosing the wrong software design pattern. Overconfidence in a solution choice also falls under this category.*
- `Time Management Errors (M07)` – *Errors resulting from a lack of time management, such as failing to allocate enough time for the implementation of a feature, procrastinating a development task, or predicting the time required for a task incorrectly.*
- `Inadequate Testing (M08)` – *Failure to implement necessary test cases, failure to consider necessary test inputs, failure to implement a certain type of testing (e.g. unit, penetration, integration) when it is necessary, or failure to consider edge cases or unexpected inputs.*
- `Incorrect/Insufficient Configuration (M09)` – *Errors in configuration of libraries/frameworks/environments or errors related to missing configuration options. Examples include misconfigured IDEs or text editors, improper directory structure for a specific programming language, missing SSH keys, missing or incorrectly named database fields or tables, missing or incorrectly named/formatted configuration files, or not installing a required library.*
- `Internationalization/String Encoding Errors (M11)` – *Errors related to internationalization and/or string/character encoding. Examples include using ASCII instead of Unicode, using UTF8 when UTF16 was necessary, failure to design the system with internationalization in mind, or failing to verify the character length of user input.*

We repeated the process of model selection outlined in Section 5.3.2.2 to see whether our improvements to T.H.E.S.E. category definitions improved classification using the models that were not selected for our ensemble classifier. The new *best* models (those with the highest median F1 scores) are summarized in Table 5.6. See Appendix F for the full results of our repeated model selection.

5.3.3 Results

We present our results for **RQ10** and **RQ11** in this section. For **RQ11**, when we discuss *median* precision, recall, and F1, we are referring to the *median of all 21 Sentence-BERT models across all experiments.*

5.3.3.1 Category Ambiguity

RQ 10: How well can ambiguity of T.H.E.S.E. category descriptions be reduced based on semantic similarity?

By examining classification results from an ensemble of Sentence-BERT classifiers, we were able to reduce ambiguity of T.H.E.S.E. category definitions by adding examples from CWE and HET [12]. Improving category definitions led to increased precision (+0.073, +0.057) and F1 (+0.068, +0.037) for classification of slips and mistakes, but a decrease in precision (-0.021) and F1 (-0.032) for lapses.

To address T.H.E.S.E. category ambiguity, we made improvements to category definitions in three ways: (1) adding examples for categories with high false positive and false negative rates, (2) adding extended descriptions from related CWE entries, and (3) adding extended descriptions to general slip, lapse, and mistake categories from the HET [12]. After each improvement attempt, we re-ran classification using our ensemble of selected Sentence-BERT models. Classification results for each attempt are summarized in Table 5.9.

This first round of improvements resulted in a slight increase in F1 for slips, but a slight decrease in F1 for lapses and mistakes. Notably, false positives decreased for lapses and mistakes, and false negatives decreased for slips and mistakes. The second round of improvements resulted in increased precision and F1 for slips and mistakes, with continued decrease in precision and F1 for lapses. The third attempt resulted in a slight increase in precision and F1 for all three human error types, but precision and F1 for lapses remained lower than before we attempted any improvements.

Attempts 1 and 2 saw a decline in precision and F1 for lapses, while precision and F1 for slips and mistakes increased. However, improving T.H.E.S.E. category definitions resulted in increased average precision, recall, and F1 scores for our ensemble classifier, as well as a small increase in average true positives and true negatives, and a small decrease in average false positives and false negatives. These results indicate that adding more relevant natural language to the descriptions for slips and mistakes can improve classification (*i.e.* reduce ambiguity in T.H.E.S.E. category definitions) using cosine similarity from Sentence-BERT models and cosine similarity, however without more examples of lapses in our dataset, classification of lapses is not likely to improve.

We find it interesting that CWE entries typically cover human errors related to implementation (*e.g.* `Code Logic Errors (M01)`, `Forgetting to Remove Development Artifacts (L03)`), but not human errors related to the general process of software engineering (*e.g.* `Internal Communication Errors (M04)`, `Overlooking Proposed Code Changes (S06)`). This may indicate a knowledge gap in existing computer security taxonomies.

5.3.3.2 Assisted Categorization

RQ 11: How useful is Sentence-BERT for refining human error definitions?

Some pretrained Sentence-BERT models can be used to automatically classify human errors as slips and mistakes with decent precision (maximum: 67.6% and 94.7%) and F1 (maximum: 57.7% and 71.8%), but precision (maximum: 28.1%) and F1 (maximum: 39.0%) for lapses is poor. Some models perform better than others, but no model is perfect. Ignoring apology comments generally increased precision and F1 for all three human error types. Our results indicate that an ensemble of three classifiers—one each for slips, lapses, and mistakes—yields more stable classifications.

Before improving T.H.E.S.E. category definitions, the best models were SB03 for slips, SB12 for lapses, and SB09 for mistakes, with median F1 scores of 0.455, 0.318, and 0.638, respectively. Our ensemble of these models yielded overall average precision, recall, and F1 of 0.435, 0.473, and 0.381, respectively. After improving T.H.E.S.E. category definitions, SB03's median F1 rose to 0.475, while median F1 decreased for SB12 (0.195) and SB09 (0.502). However, our ensemble classifier's average precision, recall, and F1 all improved (0.471, 0.481, 0.405, respectively), despite the decrease in precision and F1 for lapses.

Mistakes were the easiest type of human error to classify, with most models having a precision over 60% for mistakes, and some with exceptional precision for mistakes (94.7% for multiclass classification without apologies and without preprocessing for SB04). This is likely due to mistakes making up the majority (61.1%) of our dataset. Clearly, lapses are difficult to classify, most likely because they are underrepresented in our dataset (12.8% of the human error descriptions).

We conducted eight classification experiments varying by three factors: (1) classification with and without apology comments, (2) classification with and without preprocessing, and (3) multiclass vs. binary classification. Generally, ignoring apology comments resulted in increased precision and F1 for all three human error types. Results are tabulated in Appendix F and summarized below:

- **Impact of Ignoring Apologies on F1:** For binary classification without preprocessing, ignoring apologies resulted in increased average F1 (range: +[0.020, 0.057]) for slips, lapses, and mistakes. However, for binary classification with preprocessing, we saw the opposite: decreased average F1 (range: -[0.045, 0.059]) for slips, lapses, and mistakes. Results were similar for multiclass classification: ignoring apologies without preprocessing resulted in increased average F1 (range: +[0.024, 0.056]) for all three human error types; ignoring apologies with preprocessing resulted in increased average F1 for slips (+0.007) and mistakes (+0.191), but decreased average F1 for lapses (-0.003).

- **Impact of Ignoring Apologies on Precision:** For multiclass classification, regardless of preprocessing, ignoring apology comments resulted in increased average precision for slips (range: +[0.074, 0.126]) and lapses (range: +[0.042, 0.044]), and decreased average precision for mistakes (range: -[0.015, 0.019]). For binary classification without preprocessing, ignoring apology comments resulted in increased average precision for slips (+0.090) and lapses (+0.041), and decreased average precision for mistakes (-0.008). For binary classification with preprocessing, ignoring apology comments resulted in decreased average precision for slips (-0.039), lapses (-0.025), and mistakes (-0.100).

- **Impact of Preprocessing on F1:** For multiclass classification, regardless of whether we ignored apology comments, preprocessing resulted in an increase in F1 for mistakes (range: +[0.188, 0.191]), but mixed results for slips (decrease of -0.015 with apologies, increase of +0.007 without apologies) and lapses (increase of +0.001 with apologies, decrease of -0.003 without apologies). For binary classification, preprocessing resulted in decreased F1 for lapses (range: -[0.004, 0.105]), increased F1 for mistakes (range: +[0.148, 0.213]), and mixed results for slips (increase of +0.036 with apologies, decrease of -0.079 without apologies).

- **Impact of Preprocessing on Precision:** Regardless of whether we ignored apology comments, preprocessing comments for binary classification resulted in decreased average precision for slips (range: -[0.024, 0.153]), lapses (range: -[0.007, 0.073]), and mistakes (range: -[0.014, 0.106]). For multiclass classification, preprocessing generally resulted in increased average precision for slips (-0.026 decrease with apologies; +0.025 increase without apologies), lapses (range: +[0.010, 0.011]), and mistakes (range: +[0.026, 0.030]).

- **Impact of Multiclass vs. Binary on F1:** Binary classification without preprocessing resulted in increased F1 for slips (range: +[0.011, 0.017]), decreased F1 for mistakes (range: -[0.019, 0.023]), and mixed results for lapses (increase of +0.002 with apologies, decrease of -0.001 without apologies) over multiclass classification. Binary classification with preprocessing resulted in decreased F1 for lapses (range: -[0.004, 0.103]), and mixed results for slips (increase of +0.062 with apologies, decrease of -0.069 without apologies) and mistakes (increase of +0.006 with apologies, decrease of -0.067 without apologies) over multiclass classification.

- **Impact of Multiclass vs. Binary on Precision:** Binary classification with preprocessing, regardless of whether we ignored apology comments, resulted in decreased average precision for slips (range: -[0.010, 0.174]), lapses (range: -[0.016, 0.084]), and mistakes (range: -[0.027, 0.108]) than for multiclass classification. Binary classification without preprocessing generally resulted in increased average precision for slips (-0.012 decrease with apologies; +0.004 increase without apologies, lapses (range: +[0.000, 0.001]), and mistakes (range: +[0.017, 0.024]) over multiclass classification.

We note that our data is based on descriptions of software defects, not on descriptions of the defect in the context of human error. As established in Chapter 1, the terminology of human error is not widely used in software engineering today. Thus, our ensemble classification results are promising, suggesting that in some cases, the semantic content of a defect description can provide insight into the underlying human error that resulted in the defect.

5.3.4 Limitations

A notable aspect of this experiment is that we did not train any models, we simply evaluated pretrained models. Training would be a considerable challenge since our dataset is small (368 data points) and imbalanced (26.1% slips, 12.8% lapses, and 61.2% mistakes). We could, in theory, manually categorize more human error descriptions from our GitHub dataset, but the time required to do so may not be worth it—recall that our goal is not a perfect classifier, our goal is to use cosine similarity from Sentence-BERT models to reduce ambiguity in T.H.E.S.E. category definitions. In Section 6.2, we present a proof-of-concept for our human error reflection process that uses our ensemble classifier to provide a suggested human error type and T.H.E.S.E. category; here, too, a perfect model is not necessary. The goal of human error reflection is to have software engineers confront and organize their human errors. In this scenario, a classifier that is typically correct could encourage lazy reflection, whereas a classifier that is sometimes wrong encourages more active reflection. The classifier for our proof-of-concept is an aid for humans, not a replacement; replacing humans is the old view of human error [76], instead we want to assist software engineers experiencing human errors in evaluating what went wrong and what can be improved in the future. Another limitation is the seeming lack of Sentence-BERT models trained for software engineering tasks. There are a variety of BERT models tailored to software engineering tasks, notably sentiment analysis [33, 45], but standard BERT models are not suitable for semantic similarity of sentences or paragraphs of text. Sentence-BERT models were designed for tasks such as that, but we could not find any usable[6] Sentence-BERT models tailored to software engineering text. We also could not find any Sentence-BERT models trained on human error language.

5.3.5 Summary

In this study, our goal was to (1) reduce ambiguity in T.H.E.S.E. category definitions and (2) evaluate the feasibility of human error classification with Sentence-BERT models. To that end, we evaluated 21 pretrained Sentence-BERT

[6]We found one Sentence-BERT model [73] trained on software engineering requirements documents, but the released model is missing the configuration details needed to use it. Even if we could have tested this model, requirements engineering is only one phase of software engineering.

models to identify the best models for classifying slips, lapses, and mistakes, respectively. We implemented an ensemble classifier based on those best models which initially yielded average precision, recall, and F1 scores of 0.435, 0.473, and 0.381, respectively. We examined false positive and false negative classifications from our ensemble classifier to improve T.H.E.S.E. category definitions (*i.e.* reduce definition ambiguity) by adding our own examples and creating extended descriptions from CWE entries and human error categories in the HET [12]. With these improvements, our ensemble classifier had improved average precision (0.471), recall (0.481), and F1 (0.405). Our results indicate that some Sentence-BERT models can classify slips and mistakes, but our dataset does not have enough examples of lapses to conclude the feasibility of lapse classification. The key contributions of this research are as follows:

- Demonstration of an ensemble classifier (based on pretrained Sentence-BERT models) for human error classification
- Improved definitions and extended descriptions for T.H.E.S.E. categories

Chapter 6

Human Error Informed Micro Post-Mortems

In this chapter, we outline a formal human error informed micro post-mortem process to accompany T.H.E.S.E. and describe a proof-of-concept GitHub workflow that facilitates human error reflection using T.H.E.S.E. We conclude this chapter with some illustrative examples of our vision for human error informed micro post-mortems in practice.

6.1 Human Error Reflection Process

6.1.1 Motivation

Software engineers are familiar with a variety of post-mortem activities (*e.g.* root cause analysis), but typical post-mortems in software engineering focus on software defects. Behind every software fault and failure lies a human error [349]. Human error assessment, a form of post-mortem, has reduced the incidence and impact of accidents in the medical [98, 159, 305] and transportation [300, 332] domains. In keeping with our goal **to help software engineers confront and reflect on their human errors**, we designed a formal human error informed micro post-mortem process inspired by existing software engineering post-mortem processes [14, 40, 66, 81, 85, 162, 331, 359], human reliability analysis [4, 79, 134, 193, 335, 335], and after-event reviews [10, 86, 120]. We designed this process keeping five shared aspects of software engineering post-mortems (identified by Dingsøyr [80]) in mind: (1) select relevant participants, (2) identify what went well and why, (3) identify what did not go well, including challenges faced, (4) identify improvements for future software development, and (5) document the post-mortem. This process closely matches the process that participants followed throughout our user study (Section 5.2), incorporates feedback from our user study, and adds team-based reflection in the final step. The full process is shown visually in Figure 6.1. Throughout this section, we use the term **reviewer** to refer to the software engineer currently reflecting on their human error.

6.1.2 Step 1: Summarize Defect

The first step of the human error reflection process is to summarize the defect (*i.e.* fault or failure) that occurred. This step ensures that the reviewer spends some time reflecting on the defect before attempting to categorize their human error. Some example defect summaries from our user study are included below:

> **Participant 1**: *"There's an offset value that needs to be accounted for when drawing stuff on the matrix. So far, I've fixed this with each component individually rather than making an effort each time something new is drawn to add padding for this offset. Right now it results in messed up sidebar code that goes too high."*

> **Participant 2**: *"Forgetting to specify the remote server name in my SQL query calls (I tried "select * from listeners" instead of "select * from server.listeners")"*

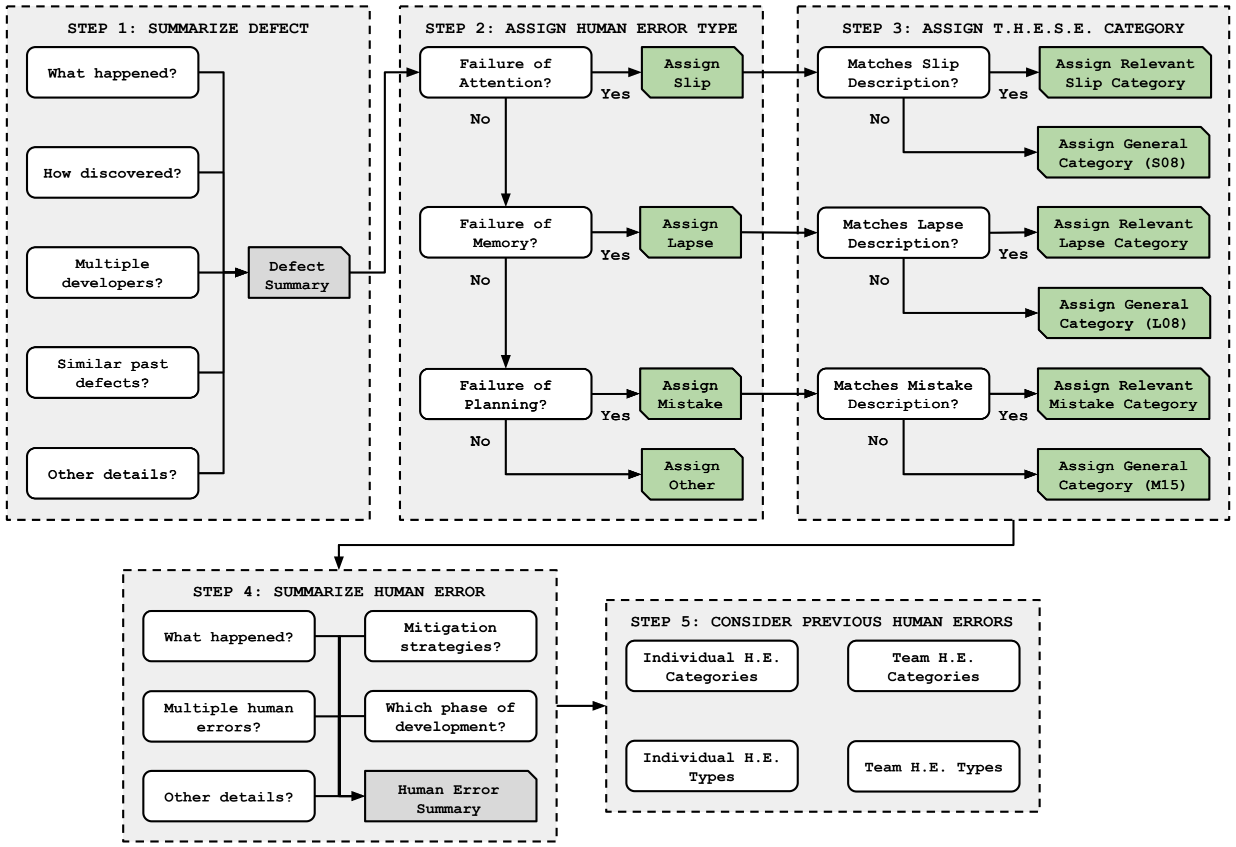

STEP 1: SUMMARIZE DEFECT
What happened?
How discovered?
Multiple developers?
Similar past defects?
Other details?
Defect Summary
STEP 2: ASSIGN HUMAN ERROR TYPE
Failure of Attention?
Yes
Assign Slip
No
Failure of Memory?
Yes
Assign Lapse
No
Failure of Planning?
Yes
Assign Mistake
No
Assign Other
STEP 3: ASSIGN T.H.E.S.E. CATEGORY
Matches Slip Description?
Yes
Assign Relevant Slip Category
No
Assign General Category (S08)
Matches Lapse Description?
Yes
Assign Relevant Lapse Category
No
Assign General Category (L08)
Matches Mistake Description?
Yes
Assign Relevant Mistake Category
No
Assign General Category (M15)
STEP 4: SUMMARIZE HUMAN ERROR
What happened?
Multiple human errors?
Other details?
Mitigation strategies?
Which phase of development?
Human Error Summary
STEP 5: CONSIDER PREVIOUS HUMAN ERRORS
Individual H.E. Categories
Team H.E. Categories
Individual H.E. Types
Team H.E. Types

Figure 6.1: Visual Summary of Our Human Error Reflection Process

Categories of slips, lapses, and mistakes are outlined in Table 4.6 and Table 4.16. Key: H.E.—Human Error.

Participant 3: *"My team went ahead and planned out state/process diagrams before meeting with the sponsor. After meeting them, we realized that the requirements given in the description did not match requirements given from the sponsor directly."*

Participant 4: *"Did not implement a necessary function for Drag and Drop API and was unable to drag an element."*

Participant 5: *"Had my code directory structured wrong so I couldn't import modules."*

We see from these examples that software developers provide different levels of detail when reporting defects. We encourage any software engineers adopting our human error reflection process to be as detailed as necessary, without providing so much detail that they could confuse themselves or their peers. Some recommended questions to consider when summarizing the defect include:

- Did actions from multiple software developers lead to the defect?
- How was the defect discovered?
- How is the defect related to other defects?

6.1.3 Step 2: Assign Human Error Type

The next step is to determine whether the defect described in Step 1 resulted from a failure of attention (*i.e.* slip), a failure of memory (*i.e.* lapse), or a planning failure (*i.e.* mistake). We encourage the reviewer to have the definitions and examples of slips, lapses, and mistakes readily available when making this determination. Some questions that may aid in assigning a human error type include:

- During which phase of software engineering did the human error occur?
- Did the error occur while following the steps of a plan? If yes, Anu *et al.* [14] would consider this a mistake.
- Did the error occur because the software engineer performed a planned step incorrectly? If yes, Anu *et al.* [14] would consider this a slip. If no, Anu *et al.* [14] would consider this a lapse.

Some example assignments from our user study:

> **Participant 1:** *"Although there have been some big changes to VHP, my matrix branch is still very much behind some of the newer code, even though I planned to change that when I had more time. I just haven't gotten around to it yet."* → **Lapse**
>
> **Participant 2:** *"Whitespace was being put in a JS template literal by VS Code's "Prettier" extension, I kept removing the white space but it would re-add it every time I saved. Out of confusion, I started looking for the solution in other places."* → **Mistake**
>
> **Participant 3:** *"Had a miscommunication with team members in regards to what the team website should include."* → **Mistake**
>
> **Participant 4:** *"Wrote arr[3] instead of arr[2] and got out of bounds when running code. Was tired and I put 3 because it was the third element."* → **Slip**
>
> **Participant 5:** *"Accidentally made two instances of a class, causing serial port access errors."* → **Mistake**

If the defect cannot be assigned a human error type, we recommend that the reviewer revisit Step 1 and provide more details on the defect in question. If the reviewer truly cannot make a distinction between slip, lapse, and mistake for the defect in question, it is likely that the reviewer experienced a symphony of human errors leading to the defect; the reviewer should choose the most relevant human error type. While T.H.E.S.E. does contain an *Other* category, we do not recommend using it, since the goal is human error reflection.

6.1.4 Step 3: Assign T.H.E.S.E. Category

After determining whether they experienced a slip, lapse, or mistake, the reviewer should carefully examine the associated human error categories and identify the category that most closely matches the human error that they experienced. If none of the specific categories match what occurred, the reviewer should select the general category for the corresponding human error type (*i.e.* `S08`, `L08`, `M15`, for general slips, lapses, and mistakes, respectively). Some example categorizations from our user study:

> **Participant 1:** *"Although there have been some big changes to VHP, my matrix branch is still very much behind some of the newer code, even though I planned to change that when I had more time. I just haven't gotten around to it yet."* → **Lapse** → `Working With Outdated Source Code (L04)`
>
> **Participant 2:** *"Whitespace was being put in a JS template literal by VS Code's "Prettier" extension, I kept removing the white space but it would re-add it every time I saved. Out of confusion, I started looking for the solution in other places."* → **Mistake** → `Incorrect/Insufficient Configuration (M09)`
>
> **Participant 3:** *"Had a miscommunication with team members in regards to what the team website should include."* → **Mistake** → `Internal Communication Errors (M04)`
>
> **Participant 4:** *"Wrote arr[3] instead of arr[2] and got out of bounds when running code. Was tired and I put 3 because it was the third element."* → **Slip** → `General Attentional Failure (S08)`
>
> **Participant 5:** *"Accidentally made two instances of a class, causing serial port access errors."* → **Mistake** → `Code Logic Errors (M01)`

6.1.5 Step 4: Summarize Human Error

After categorizing their human error, the reviewer should summarize their human error. This summary should go beyond the defect description and discuss the human error that led to the defect. Some examples corresponding to the examples from Section 6.1.2 are included below:

> **Participant 1:** *"I thought that I accounted for that when I did the whole initial– make the lines of code and color them, but then I noticed on some files, the lines will extend almost above the darker area. I knew it was an issue in the past, so I think if I had just like set a hard value to it– because I'm– my idea on it now is when I find it, it's gonna be because whatever part that I'm drawing, that line of code is not accounting for that amount of padding I need to give it. So I think if I just went ahead before and said 'alright, I know this is probably gonna be an issue'– if I just add a variable called code_padding with a value of like 50 or something, and then add it to stuff I know is gonna get drawn in the code field, then I can like solve it for the future when I try to mess around with it."*
>
> **Participant 2:** *"I should have looked at one of our stored procedures before I tried writing it. I usually look at another query or two before I start writing ours just so I can remember what relation names are and stuff like that, so just looking at an example for the query would have been helpful. I definitely wouldn't have made the mistake if I looked at an example."*
>
> **Participant 3** had little more to say about this human error.
>
> **Participant 4** indicated that they had never worked with a drag-and-drop workflow and didn't know they needed a function to handle that.
>
> **Participant 5:** *"It's not really a configuration thing, more of me forgetting that python has a specific structure for importing modules from the same project and not designing my code around that, then flailing around trying to fix it."*

We encourage the reviewer to consider the following questions when summarizing their human errors:

- Were there multiple human errors that led to this defect? If so, what order did they occur in?
- What mitigation strategies (*e.g.* tools, techniques, processes) could help prevent similar human errors in the future?
- During which phase of software engineering did this human error occur?

If the reviewer chose a general category (*i.e.* `S08`, `L08`, `M15`, `Other`), then they should also discuss the following:

- What makes this human error different? Why does it not fit in an existing category?
- What would a fitting human error category be called? What other defects could fit within this new category?

6.1.6 Step 5: Consider Previous Human Errors

After providing relevant details about the specific human error, the reviewer should consider their human error in the following contexts:

Context 1: **Individual Human Error Categories:**
Has the reviewer experienced human errors in this T.H.E.S.E. category in the past? If so:

- How is the current human error similar to previous ones?
- Were the mitigations suggested for similar previous human errors implemented? If so, why didn't they prevent the current human error? If not, are they still relevant mitigations, or should alternative mitigation strategies be implemented?

Context 2: **Individual Human Error Types:**
Has the reviewer experienced the same type of human error (*i.e.* slip, lapse, mistake) in the past? If so:

- How often does the reviewer experience this type of human error compared to the other types?

- Is this type of human error being experienced more or less frequently? Why?

Context 3: **Team Human Error Categories:**
What T.H.E.S.E. categories are frequently experienced by developers on the reviewer's team? Consider the following:

- For the most frequent categories of human error, why have the suggested mitigations failed to prevent human errors in those categories?
- Why do infrequent categories of human error occur less often? Are there mitigations in place to prevent them? If so, why have they been effective?

Context 4: **Team Human Error Types:**
What types of human errors (*i.e.* slips, lapses, mistakes) are experienced most often by the reviewer's team? Consider the following:

- For the most frequent human error types, why do they occur so often?
- For infrequent human error types, why do they rarely occur?

We encourage software engineering teams to discuss Context 3 and Context 4 as a group. These contexts will be of particular interest to managers of software engineering teams, as they analyze human error trends within a software project and/or team.

6.1.7 Summary

In this section, we formally describe our human error informed micro post-mortem process, which utilizes T.H.E.S.E. for human error reflection. This process is intended to be general to all phases of software engineering and simple to follow. Our human error reflection process was inspired by existing software engineering post-mortem processes [14, 40, 66, 81, 85, 162, 331, 359], human reliability analysis [4, 79, 134, 193, 335, 335], and after-event reviews [10, 86, 120] and encompasses five steps: (1) summarize defect, (2) assign human error type, (3) assign T.H.E.S.E. category, (4) summarize human error, and (5) consider previous human errors. Steps 1-4 of our process were tested and evaluated in our user study (Section 5.2), where participants agreed that human error reflection with T.H.E.S.E. is a beneficial software engineering activity.

6.2 Proof-of-Concept: Human Error Reflection Engine (H.E.R.E.)

6.2.1 Motivation

One challenge for adoption of new processes is ease of use [74, 343]. Since software engineers are indeed people, it follows that software engineers are not likely to use our human error reflection process without first seeing its benefit.

We implemented the Human Error Reflection Engine (H.E.R.E.), a proof-of-concept GitHub workflow, to lower the barrier to entry and give software engineers a chance to experience human error reflection with minimal setup time and effort. This section outlines how H.E.R.E. was implemented, how to deploy H.E.R.E. on a GitHub repository, and how software engineers can interact with H.E.R.E. Figure 6.2 shows visually what actions H.E.R.E. takes in response to interaction from a software engineer.

6.2.2 Reviewer Experience

To interact with H.E.R.E., the reviewer (*i.e.* the software developer engaging in human error reflection) first needs to opt-in by applying the `to-err-is-human` label to an issue on their repository. Upon doing so, H.E.R.E. feeds natural language—from the issue's description and comments—to our Sentence-BERT-based classifier (the final version with extended descriptions) as described in Section 5.3. The classifier responds to H.E.R.E. with a recommended human error type and category. H.E.R.E. takes the recommendation and inserts it into a comment template, which is then posted on the issue. An example of the comment template is included in Figure 6.3.

The reviewer then reflects on their human error by examining the recommended human error type and category, and determining the human error type (as described in Section 6.1.3). Next, the reviewer categorizes their human error (as described in Section 6.1.4) and selects one or more of the check boxes in the H.E.R.E. comment corresponding with their T.H.E.S.E. categorization. H.E.R.E. then applies the relevant labels to the issue. When the reviewer is

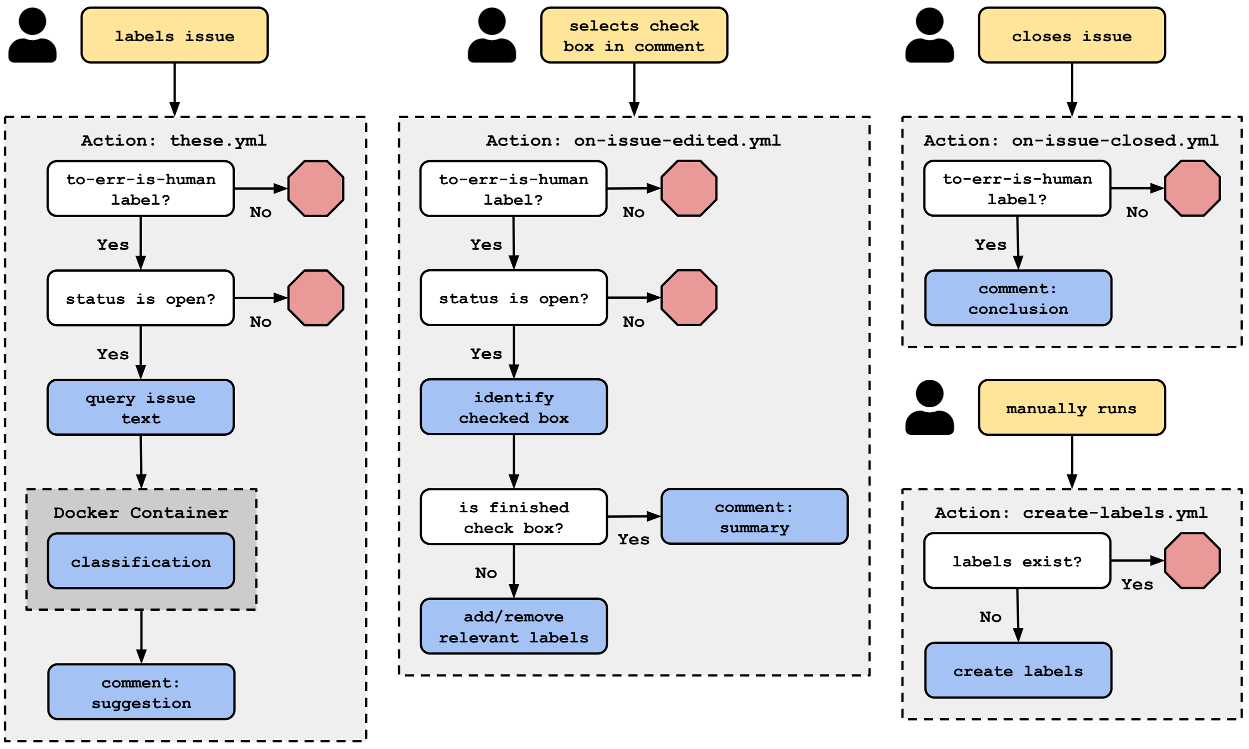


Figure 6.2: Human Error Reflection Engine (H.E.R.E.) Workflow

Key: Yellow—developer action; White—conditional check; Blue—automated action; Red—do nothing.

finished categorizing, they select the `Finished` check box, and H.E.R.E. posts a new comment with a summary of the reviewer's categorization and some self-reflection questions. An example of this comment is included in Figure 6.4.

The reviewer is encouraged to respond with a comment addressing the self-reflection questions (Step 4 of our human error reflection process, as described in Section 6.1.5). Finally, when the reviewer closes the issue, H.E.R.E. posts a final comment, congratulating the reviewer for reflecting on their human error and reminding them how to start this process on future issues. An example of this concluding comment is included in Figure 6.5.

6.2.3 Implementation Details

H.E.R.E. consists of four GitHub Actions scripts and a Docker container with our Sentence-BERT-based classifier (described in Section 5.3). The code is available on GitHub[1]. The four GitHub Actions scripts that compose H.E.R.E. are described as follows:

- `create-labels.yml`: This action creates the labels that H.E.R.E. uses to organize software developers' human errors within a repository. Labels created include a special `to-err-is-human` label that triggers `these.yml` to run, as well as labels corresponding to each human error type and each T.H.E.S.E. category. If the labels already exist, they are updated. This action must be manually executed by a developer with the appropriate permissions to do so. This action uses the GitHub CLI[2], which is pre-installed for all GitHub Actions.

- `these.yml`: This action is triggered when the `to-err-is-human` label is applied to a GitHub issue. Once triggered, this action uses GitHub's GraphQL API to collect the natural language from the issue description and comments. This action sends the natural language to our Sentence-BERT-based classifier (inside a Docker container). When the classifier responds with a recommended human error type and category, this action comments on the issue with the comment shown in Figure 6.3.

- `on-issue-edited.yml`: This action is triggered when the reviewer selects/unselects a check box on the comment shown in Figure 6.3. When triggered, this action adds/removes the corresponding human error type/T.H.E.S.E. category labels. If the special `Finished` check box is selected, this action posts the comment shown in Figure 6.4.

[1]`https://github.com/meyersbs/these-poc`
[2]`https://cli.github.com/`

Hello, there! I'm the **Human Error Reflection Engine (H.E.R.E.)**. My purpose is to help you document and reflect on your **Human Errors**, actions that result in something that was

> *"not intended by the actor; not desired by a set of rules or an external observer; or that led the task or system outside its acceptable limits [Source]."*

In other words, human errors are actions that lead to unintended, unexpected, or undesirable outcomes.

Don't be shy, everyone experiences human errors, and I'm not here to judge. I just want to help you learn from your human errors, so, let's get started!

Step 0: My Assessment

Based on the natural language description of this issue, I suspect your human error is: `<CLASSIFIER_RESULTS>`

Don't worry, that's just my best guess. If that's wrong, you can use the next steps to determine what actually happened.

Step 1: Slip, Lapses, or Mistake?

There are three types of human error that we are concerned with:

- **Slips:** Failing to complete a properly planned step due to inattention, such as putting the wrong key in the ignition, or overlooking stakeholder requirements.
- **Lapses:** Failing to complete a properly planned step due to memory failure, such as forgetting to put the car in reverse before backing up, or forgetting to check if a pointer is non-null before dereferencing it.
- **Mistakes:** Planning errors that occur when the plan is inadequate, such as getting stuck in traffic because you didn't consider the impact of the bridge closing, or choosing an inadequate sorting algorithm.

Alright, now that you understand slips, lapses, and mistakes, let's label your human error. Start by deciding if this issue resulted from a slip, lapse, or mistake. Once you have determined that, move on to Step 2.

Step 2: Assign Human Error Category

Now that you've determined whether your human error was a slip, lapse, or mistake, select the human error categories below that best describe what happened.

Slips (Attentional Failures)

- ☐ S01 **Typos & Misspellings**: Typos and misspellings may occur in code comments, documentation (and other development artifacts), or when typing the name of a variable, function, or class. Examples include misspelling a variable name, writing down the wrong number/name/word during requirements elicitation, referencing the wrong function in a code comment, and inconsistent whitespace (that does not result in a syntax error).
- ☐ • • •

Lapses (Memory Failures)

- ☐ L01 **Forgetting to Finish a Development Task**: Forgetting to finish a development task. Examples include forgetting to implement a required feature, forgetting to finish a user story, and forgetting to deploy a security patch.
- ☐ • • •

Mistakes (Planning Failures)

- ☐ M01 **Code Logic Errors**: A code logic error is one in which the code executes (*i.e.* actually runs), but produces an incorrect output/behavior due to incorrect logic. Examples include using incorrect operators (*e.g.* += instead of +), erroneous if/else statements, incorrect variable initializations, problems with variable scope, and omission of necessary logic.
- ☐ • • •

Other

- ☐ Other: Only use this category if none of the other categories describe your error.

Step 3: Finished Categorizing

When you are finished categorizing (checking boxes above), please check the following box:

- ☐ Finished

Notes

- In **Step 0**, H.E.R.E. uses natural language processing (cosine similarity with sentence-BERT) to try and categorize your human error for you. This is an experimental feature and should be verified.
- The human error types in **Step 1** come from James Reason's Generic Error-Modelling System (GEMs). You can read more about slips, lapses, and mistakes here.
- The specific categories of human error in **Step 2** come from the Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.). You can read more about T.H.E.S.E. here.

Figure 6.3: H.E.R.E. Comment to Guide Human Error Categorization

Key: `<CLASSIFIER_RESULTS>`—suggested human error type and category; • • •—truncated list of T.H.E.S.E. categories; Blue—hyperlink.

You experienced the following human errors:

> `<LIST_OF_LABELS>`

To get the most out of this human error assessment, please take some time to comment below with more details, such as:

- What went wrong?
- If you selected a general category of human error, why doesn't it fit with an existing category and what would you call it?
- How did each human error that you experienced manifest in the source code?
- How can you avoid similar human errors in the future?

Figure 6.4: H.E.R.E. Comment to Guide Human Error Reflection
Key: `<LIST_OF_LABELS>`—list of human error labels applied to this issue.

Congratulations on assessing your human errors!
If you wish to keep using the **Human Error Reflection Engine (H.E.R.E.)**, just add the `to-err-is-human` label to any issue to get started.

> *"Mistakes are the portals of discovery."* – James Joyce

> *"The only real mistake is the one from which we learn nothing."* – Henry Ford

Figure 6.5: H.E.R.E. Comment to Conclude Human Error Reflection
This comment reminds software engineers how to enable H.E.R.E. and provides some reassuring quotes about human error.

- `on-issue-closed.yml`: This action is triggered when the issue is closed. When triggered, this action posts the comment shown in Figure 6.5.

These actions make use of the following open source actions:

- `octokit/graphql-action`[3] to query GitHub's GraphQL API
- `thollander/actions-comment-pull-request`[4] to comment on the issue
- `actions-ecosystem/action-regex-match`[5] to identify which check boxes are selected

Our Docker container is relatively simple, consisting of a single python script (our Sentence-BERT classifier) and the python libraries required to run the script. The Dockerfile defining our container is available on GitHub [6]. We have an additional GitHub action, `docker-image.yml`[7], which builds our container and pushes it to DockerHub[8] for easy access from `these.yml`.

6.2.4 Deployment Process

In this section, we describe the process for deploying H.E.R.E. in a GitHub repository. The process outlined below assumes the reader has created a valid GitHub repository. The reader must be the owner of the repository to follow this process. Replace `<OWNER>` or `<REPO>` below with the owner of the repository (*i.e.* the reader's GitHub username) and the name of the repository, respectively.

Step 1: **Configuration**

(a) Go to `https://github.com/<OWNER>/<REPO>/settings/actions`
(b) Under the `Workflow permissions` section, select the `Read and write permissions` option
(c) Click the `Save` button

[3]`https://github.com/octokit/graphql-action`
[4]`https://github.com/thollander/actions-comment-pull-request`
[5]`https://github.com/actions-ecosystem/action-regex-match`
[6]`https://github.com/meyersbs/these-poc/blob/main/Dockerfile`
[7]`https://github.com/meyersbs/these-poc/blob/main/.github/workflows/docker-image.yml`
[8]`https://hub.docker.com/repository/docker/meyersbs/these-poc/general`

Step 2: **Setup**

(a) In the repository, ensure that the following path exists: `.github/workflows/`

(b) Copy four `.yml` files from `https://github.com/meyersbs/these-poc/tree/main/.github/workflows` into `.github/workflows/` with the following modifications:

- `create-labels.yml` Line 10: Replace `meyersbs/these-poc` with `<OWNER>/<REPO>`
- `these.yml` Line 42: Replace `meyersbs` with `<OWNER>`
- `on-issue-edited.yml` Line 721: Replace `meyersbs` with `<OWNER>`
- `on-issue-closed.yml`: No changes necessary

(c) Commit and push these files to your repository

(d) Go to `https://github.com/<OWNER>/<REPO>/actions/workflows/create-labels.yml`

(e) Click the `Run workflow` dropdown on the right side of the screen

(f) Click the green `Run workflow` button to automatically create the necessary issue labels

The remaining GitHub actions (`these.yml`, `on-issue-edited.yml`, `on-issue-closed.yml`) will run automatically when the necessary conditions are met on an issue in your repository. The Dockerfile and `docker-image.yml` workflow can be disregarded by software engineers deploying H.E.R.E.

6.2.5 Limitations

H.E.R.E. does not facilitate team-based human error reflection (Context 3 and Context 4 in Section 6.1.6) directly, however GitHub does provide a way to view all issues with a specific label, so software engineers (and their managers) can easily access lists of all of the slips (for example) reported on a repository. Additionally, H.E.R.E. has no way to force the reviewer to participate in Step 4 of our human error reflection process (as described in Section 6.1.5). However, a software engineer opting to use H.E.R.E. is unlikely to ignore the process.

The workflow in `these.yml` takes about three minutes to execute. This is primarily due to how GitHub Actions handles loading/caching Docker containers. Software development teams adopting H.E.R.E. into their post-mortem activities could, in theory, speed this process up by using a self-hosted runner[9].

6.2.6 Summary

The Human Error Reflection Engine (H.E.R.E.) is a proof-of-concept workflow that facilitates human error informed micro post-mortems on GitHub. We implemented H.E.R.E. with a series of GitHub actions and a Docker container with a Sentence-BERT based classifier. The intent of H.E.R.E. is to demonstrate the feasibility of our human error reflection process while also lowering the barrier to entry for software engineers who wish to adopt human error reflection with T.H.E.S.E.

6.3 Follow-Up Survey

We sent a follow-up survey (see Appendix I) to our user study participants with an accompanying slide deck (see Appendix H) outlining our formal human error informed micro post-mortem process and presenting H.E.R.E. One participant from Phase 2 (anonymously) responded, indicating, almost 7 months after completing Phase 2, that they (1) have continued thinking about their software defects in terms of slips, lapses, and mistakes, as well as T.H.E.S.E. categories, and (2) find T.H.E.S.E. and human error reflection valuable in their software engineering activities.

In regards to our human error informed micro post-mortem process, the participant indicated that the process (1) is clear and general to all phases of software engineering, (2) would lead to meaningful reflection of their human errors, and (3) would be beneficial to both software engineering teams and individual software engineers. The participant indicated that H.E.R.E. would make it easy for software engineers to adopt human error reflection assisted by T.H.E.S.E. and gave no suggestions for improving H.E.R.E.

Full responses are provided in Appendix J. While no conclusions can be drawn from a single survey respondent, we find these results reassuring.

[9]`https://docs.github.com/en/actions/hosting-your-own-runners`

6.4 Illustrative Examples

In this section, we provide some examples that illustrate our vision for T.H.E.S.E. and H.E.R.E. in practice. Note that these examples are purely fictional; all names of individuals, companies, and software systems are products of our imagination.

6.4.1 Paul's API Calls (Slip Resulting from Workplace Distractions)

TacoSoft is a software company that creates point-of-sale systems, mobile applications, and websites for restaurants that sell tacos. Shell Taco (TacoSoft's biggest customer) has asked for a mobile app that allows customers to order food for pickup and delivery. TacoSoft assigns Michelle, a manager, to oversee a team of ten software engineers developing Shell Taco's mobile app. Paul, one of Michelle's software engineers, has been tasked with integrating the mobile app with car GPS systems, online mapping services, and services that track and report local traffic conditions. Upon finishing his implementation, Paul asks Michelle and two other members of their team to review his code. During code review, Michelle notices that Paul used deprecated API calls that will be removed at the end of the year. Michelle provides feedback to Paul, he apologizes, and he fixes his code before merging it into the production branch.

Paul's next assignment involves securing communications between the mobile app and car GPS systems. Paul identifies a well-known encryption library and begins reading through documentation to find the correct functions for his needs. Paul keeps getting distracted by his coworkers, Jennifer and Emilio, and he misses a warning in the encryption library's documentation. After completing his implementation, Michelle and Paul perform another code review. Again, Michelle notices that Paul has used an improper API call.

Noticing a pattern, Michelle calls Paul into her office for a discussion about his performance. Michelle asks Paul if he has any ideas about why he keeps making these errors, but he is genuinely unsure. Michelle does some searching online and comes across T.H.E.S.E. The next day, she calls Paul in for another meeting and shows him T.H.E.S.E. Following our human error reflection process, Michelle and Paul identify that Paul's errors are slips, failures resulting from inattention. Looking closer at T.H.E.S.E., Paul realizes that he has been distracted by Jennifer and Emilio's loud conversations.

The following week, while trying to decide which function to use from a credit card processing API, Paul finds himself distracted by Jennifer and Emilio once again. Paul politely asks Jennifer and Emilio to take their conversation into the break room and they comply. Returning to his desk, Paul realizes that he initially chose the wrong API call for his needs and fixes his code. During Paul's next code review, Michelle notices that Paul did not make the same kinds of errors she had seen over the past few weeks.

This is an example of a slip, a human error resulting from inattention, due to workplace distractions. Paul's repeated slips led to faults in the mobile app source code. Using T.H.E.S.E. and its accompanying human error reflection process, Paul and Michelle were able to successfully identify the underlying human error behind the faults that Paul introduced. This process allowed Paul to confront and organize his human errors, and implement a change in his workplace environment to prevent similar human errors from occurring in the future, ultimately saving time in future implementation and debugging activities.

6.4.2 Nuthan's Neglect (Mistake Resulting from Time Constraints)

Anonymoose is a software company that develops and maintains the Anonymoose software, a mobile app that lets users send encrypted text messages to each other. Version 1.0 of Anonymoose has been out for nearly a year and has attracted a large user base. The company is working on Version 2.0 and Nuthan is tasked with speeding up the encryption/decryption process. The current implementation of Anonymoose uses RSA encryption from a well-known encryption library.

Nuthan searches online for other encryption libraries and identifies CryptQuick, which boasts significantly faster speeds than the current implementation in Anonymoose. Nuthan examines the documentation for CryptQuick, but cannot find details about the encryption algorithm; since Version 2.0's release date is quickly approaching, Nuthan assumes that CryptQuick must be using RSA. Nuthan demonstrates the improved speeds to his supervisor, Renee, and she signs off on his changes.

Version 2.0 is released and everything goes well for a few weeks. Late one night, a hacker releases a dump of 10 million user conversations from Anonymoose obtained by breaking the encryption. Renee, Nuthan, and the rest of the security team review Nuthan's changes and dig into the source code for CryptQuick. They discover that CryptQuick uses an outdated and insecure encryption algorithm. Renee works with Nuthan on a patch and releases the patch, but not before a significant cost to Anonymoose's reputation. In the months that follow, the majority of Anonymoose users switch to an alternative encrypted messaging app, and Anonymoose is not making enough money to stay in business. Anonymoose declares bankruptcy.

In this example, we demonstrate `Wrong Assumption Errors (M03)`. Specifically, we demonstrate what could happen when a software engineer is working under strict time constraints, leading them to make incorrect assumptions. Research indicates that knowledge of human errors can help prevent developers from committing those errors in the future [140, 141]. If the Anonymoose company had provided human error training to its developers, perhaps this catastrophic mistake could have been avoided. Using T.H.E.S.E. in tandem with our human error reflection process can help developers catch mistakes (and other human errors) before they can manifest as bugs and vulnerabilities in production software systems.

6.4.3 Square Corp's Snafu (Systemic Mistakes Due to Poor Requirements)

Square Corp is a software company that primarily designs and implements software systems for motor vehicles. ElectriCar, a brand new car company, asks Square Corp to implement a system that analyzes data from sensors (*e.g.* ultrasonics, accelerometers) and provides warnings to the driver when potentially unsafe actions occur (*e.g.* crossing over the white line). Square Corp assigns Richard and his team to this project. After developing a prototype, Richard's team meets with the stakeholders for a demonstration. The stakeholders are unhappy with many aspects of this prototype, including the delay between sensor readings and system responses and the type/content of warning messages provided to drivers.

Richard's team tweaks their implementation and gives another demonstration to the stakeholders, but they are still not pleased. This process repeats a few times until Richard's team produces a software product that meets the stakeholders' expectations. Upon paying for this work, ElectriCar informs Square Corp that they will be hiring a different software company for their future needs.

Square Corp's lead software engineer, Joanna, meets with Richard's team for a post-mortem assessment of what went wrong in this project, but no immediate causes are apparent. Joanna has heard from a friend at another company, Sid, about a human error informed micro post-mortem process that his team has been using. Sid recommends that Joanna enable H.E.R.E. on Square Corp's repository for the ElectriCar code. Joanna takes Sid's advice and provides some training on human error before tasking Richard's team with using H.E.R.E. for a post-mortem inspection of their errors. Richard reviews his team's responses to H.E.R.E. and sees a clear pattern in responses:

- H.E.R.E. classifies Issue #39 (Unacceptable Delay in Responses to Sensor Readings) as a *mistake*. Doug confirms, saying that *"the stakeholders never provided an expected response time during requirements elicitation and, regretfully, we did not ask them to provide clarification."*

- H.E.R.E. classifies Issue #54 (Threshold for Acceleration Warning Too Low) as a *slip*. After reviewing his notes on human error from their training, Andrew replies *"Actually, this is a mistake related to blindly following the requirements specification; the stakeholders provided an unrealistic threshold for warning that the driver is accelerating too quickly, and I implemented it exactly as documented without considering whether the threshold was reasonable. I'm sorry about that."*

- H.E.R.E. classifies Issue #67 (User Warnings are Too Specific) as a *mistake*. Beatrice confirms, adding that *"the content of warning messages was never discussed with the stakeholders during requirements elicitation. That's my fault, I should have noticed that when reviewing the requirements specification."*

Richard meets with Joanna and shows her his team's feedback. They conclude that the requirements elicitation for this project went poorly. They share their findings with Richard's team. During the next project, Richard's team continues using H.E.R.E., but Richard notices fewer errors compared to the ElectriCar project.

This example demonstrates how our human error informed micro post-mortem process can help software developers (and their managers) to identify systemic problems within the development team's requirements elicitation process by confronting and organizing their human errors.

6.5 Real World Examples

One of the original inspirations for this dissertation came from reflecting on some of the most infamous human errors that the field of software engineering has ever experienced. Vulnerabilities tend to be small errors with big consequences, and understanding how they arise in a system is critical.

In this section, we circle back to our original observations with a new perspective, examining some real-world software vulnerabilities using our human error reflection process and T.H.E.S.E. to further demonstrate our vision for T.H.E.S.E. in practice. Recall that our human error reflection process is intended to be conducted by the software engineer(s) who experienced the human error being reflected upon, so the actual software developers involved in these software vulnerabilities may arrive at different categorizations and conclusions.

6.5.1 HeartBleed (Code Logic Error Leads to Confidentiality Violations)

In 2014, a vulnerability in OpenSSL[10]—a popular library intended to provide secure communications over the internet—called HeartBleed allowed attackers to obtain sensitive information (*e.g.* usernames, passwords, emails) from process memory using specially crafted packets that triggered a buffer overflow. HeartBleed could be exploited without any credentials, allowing anyone on the internet to read memory from devices running vulnerable versions of OpenSSL. HeartBleed is documented in CVE-2014-0160, which summarizes the vulnerability:

> *"The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to `d1_both.c` and `t1_lib.c`, aka the Heartbleed bug [225]."*

So, what exactly went wrong? Conceptually, HeartBleed is quite simple: Typically, a device sends a heartbeat request to a server—a periodic request to make sure that the device can still communicate with the server. This heartbeat request includes some expected response and the length of that expected response. For example, the word *potato* and 6 characters. The server responds with the expected *potato*, which is 6 characters, and everyone is happy. With HeartBleed, attackers could craft special heartbeat requests that requested a small amount of information, but specified a large response length. Servers running vulnerable versions of OpenSSL would respond with the expected information, and then pull information from memory to fill up the remaining characters, violating the confidentiality of the system [373].

This is quite interesting, but we still don't know anything about the underlying human error. The underlying issue was a lack of bounds checking, which was fixed with about 30 lines of code [329]. We have summarized the defect, which is Step 1 of our human error reflection process. Step 2 involves determining whether the underlying human error was a slip, lapse, or mistake. We may never know the true human error type for HeartBleed, as the details of what went wrong are locked in the minds of the developers who first wrote the vulnerable code. However, if we examine T.H.E.S.E., it seems likely that this was a mistake, specifically `Code Logic Errors (M01)` (Step 3). Step 4 is to summarize the human error, which could look something like this:

> *A code logic error, specifically a lack of bounds checking, allowed remote attackers to gain confidential information from systems running vulnerable versions of OpenSSL. This human error occurred during the implementation phase of software engineering, and likely also involved `Inadequate Experience Errors (M12)`, `Wrong Assumption Errors (M03)`, and/or `General Attentional Failures (S08)`. A formal code review with experienced C programmers may have been able to mitigate this human error.*

This example demonstrates how software engineers can leverage T.H.E.S.E. to learn from well-known vulnerabilities, and potentially avoid similar human errors in the future.

[10] `https://www.openssl.org/`

6.5.2 StageFright (More to Human Error than Meets the Eye)

In 2015, a collection of vulnerabilities discovered in versions 2.2 through 5.1.1 of the Android[11] operating system—called StageFright—made it possible for attackers to execute arbitrary code on mobile devices and gain elevated privileges using specially crafted metadata in MP3 (audio) and MP4 (video) files. The exploit involved buffer overflows and a general lack of data verification. The only information the attacker needed was a phone number. StageFright is documented in multiple CVE entries:

> **CVE-2015-1538**: *"Integer overflow in the `SampleTable::setSampleToChunkParams` function in `SampleTable.cpp` in `libstagefright` in Android before 5.1.1 LMY48I allows remote attackers to execute arbitrary code via crafted atoms in MP4 data that trigger an unchecked multiplication, aka internal bug 20139950, a related issue to CVE-2015-4496 [226]."*
>
> **CVE-2015-1539**: *"Multiple integer underflows in the `ESDS::parseESDescriptor` function in `ESDS.cpp` in `libstagefright` in Android before 5.1.1 LMY48I allow remote attackers to execute arbitrary code via crafted ESDS atoms, aka internal bug 20139950, a related issue to CVE-2015-4493 [227]."*
>
> **CVE-2015-3824**: *"The `MPEG4Extractor::parseChunk` function in `MPEG4Extractor.cpp` in `libstagefright` in Android before 5.1.1 LMY48I does not properly restrict size addition, which allows remote attackers to execute arbitrary code or cause a denial of service (integer overflow and memory corruption) via a crafted MPEG-4 tx3g atom, aka internal bug 20923261 [228]."*
>
> **CVE-2015-3826**: *"The `MPEG4Extractor::parse3GPPMetaData` function in `MPEG4Extractor.cpp` in `libstagefright` in Android before 5.1.1 LMY48I does not enforce a minimum size for UTF-16 strings containing a Byte Order Mark (BOM), which allows remote attackers to cause a denial of service (integer underflow, buffer over-read, and mediaserver process crash) via crafted 3GPP metadata, aka internal bug 20923261, a related issue to CVE-2015-3828 [229]."*
>
> **CVE-2015-3827**: *"The `MPEG4Extractor::parseChunk` function in `MPEG4Extractor.cpp` in `libstagefright` in Android before 5.1.1 LMY48I does not validate the relationship between chunk sizes and skip sizes, which allows remote attackers to execute arbitrary code or cause a denial of service (integer underflow and memory corruption) via crafted MPEG-4 covr atoms, aka internal bug 20923261 [230]."*
>
> **CVE-2015-3828**: *"The `MPEG4Extractor::parse3GPPMetaData` function in `MPEG4Extractor.cpp` in `libstagefright` in Android before 5.1.1 LMY48I does not enforce a minimum size for UTF-16 strings containing a Byte Order Mark (BOM), which allows remote attackers to execute arbitrary code or cause a denial of service (integer underflow and memory corruption) via crafted 3GPP metadata, aka internal bug 20923261, a related issue to CVE-2015-3826 [231]."*
>
> **CVE-2015-3829**: *"Off-by-one error in the `MPEG4Extractor::parseChunk` function in `MPEG4Extractor.cpp` in `libstagefright` in Android before 5.1.1 LMY48I allows remote attackers to execute arbitrary code or cause a denial of service (integer overflow and memory corruption) via crafted MPEG-4 covr atoms with a size equal to `SIZE_MAX`, aka internal bug 20923261 [232]."*

Together, this symphony of software bugs created an unique opportunity for attackers. The exploit is conceptually simple: In affected versions of Android, when a text message containing an audio or video attachment is received, the media transcoding libraries automatically generate a preview of the attached file before the user even opens the text message. Using specially crafted metadata for the attached audio or video, attackers could execute arbitrary code and gain root (complete) access to the operating system on a device [125].

At first glance, it would be tempting to say that the core of this vulnerability, the buffer overflow, was caused by a software developer who either wasn't paying attention (*i.e.* slip) or forgot that they needed to consider buffer overflows (*i.e.* lapse). It may also be tempting to call this a code logic error (`M01`) and move on. These are all valid conclusions and they could have been in play, but this vulnerability isn't so simple. There was a clear design oversight, as the code listening for incoming text messages and generating previews was running as root (with full privileges). Regardless of the buffer overflows present in the transcoding libraries, the decision to allow code running as root to operate on untrusted data is a design **mistake**.

There is yet another layer of human error at play here. Once StageFright was disclosed, the fix was available quickly, but as Google tried to push the security patch to phone carriers (each with their own customized flavor of Android), software engineers procrastinated (`Time Management Errors (M07)`) porting the fix, possibly due to a lack of security knowledge (`Inadequate Experience Errors (M12)`), or to a failure on Google's part to adequately convey the severity of the vulnerability (`External Communication Errors (M05)`).

The software developers involved in StageFright might summarize their human error experience this way:

[11]`https://www.android.com/`

A collection of code logic errors (`M01`*) coupled with poor design decisions allowed attackers to gain root access to Android devices, potentially violating confidentiality, integrity, and availability of the Android operating system. This vulnerability was further exacerbated by poor communication (*`M05`*) and poor time management (*`M07`*). Diligent code review along with significant testing of edge cases and the establishment of security requirements may have been able to prevent this symphony of human errors, or at least reduce the impact of the vulnerability.*

This example demonstrates the need for careful, deep reflection on human errors in software engineering. The human error(s) behind a software defect aren't always simple to discern, and a surface-level examination of human error may lead to incorrect or incomplete conclusions and mitigation strategies.

Chapter 7

Summary & Future Work

Software engineers, despite their best efforts, experience human errors, which manifest as software defects. While software defects are routinely studied and categorized, the software engineering domain still lacks an established and accepted human error assessment process, a process which could improve the quality and security of software, as it has done in the medical [98, 159, 305] and transportation [300, 332] domains. Our goal in this dissertation is ***to help software engineers confront and reflect on their human errors by creating a process to document, organize, and analyze human errors***. To that end, this dissertation comprised three phases:

Phase 1: Systematization (*i.e.* identification and taxonomization) of software engineers' human errors from literature and development artifacts into a Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.)

Phase 2: Evaluation of T.H.E.S.E. based on software engineers' perceptions and natural language insights

Phase 3: Creation of a human error informed micro post-mortem process and the Human Error Reflection Engine (H.E.R.E.), a proof-of-concept GitHub workflow facilitating human error reflection

During Phase 1, we conducted a systematic literature review of research pertaining to human error in software engineering, which yielded Version 1 of our Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.), consisting of 12 categories of human errors experienced by software engineers spanning slips, lapses, and mistakes (see Section 4.1). Next, we manually annotated a subset of software engineers' self-admitted human errors (identified via apology lemmas) on GitHub, and discovered 15 new categories of human errors in software engineering, resulting in Version 2 of T.H.E.S.E. (see Section 4.2).

In Phase 2, we evaluated T.H.E.S.E. with a user study in which five software engineering students documented and reflected on the human errors they experienced during software development over the course of 17 weeks. Survey responses and discussions from weekly interviews led to Version 3 of T.H.E.S.E. and revealed that (1) T.H.E.S.E. has clear definitions and makes it easy to confront human errors, (2) T.H.E.S.E. covers all phases of software engineering, and (3) human error assessment assisted by T.H.E.S.E. is perceived as beneficial and meaningful to software engineers. Survey responses indicated potential ambiguity in T.H.E.S.E. category definitions, which prompted us to evaluate category definitions based on cosine similarity from pretrained Sentence-BERT models—models capable of representing the semantic content (meaning) of text. Our evaluation yielded improvements to T.H.E.S.E. category definitions (Version 4 of T.H.E.S.E.) as well as the addition of extended descriptions, while indicating that human error classification may be feasible given a larger dataset.

Finally, in Phase 3, we defined a human error informed micro post-mortem process for software engineering—which closely matches the process followed by participants in our user study—and implemented a proof-of-concept Human Error Reflection Engine (H.E.R.E.) to facilitate human error reflection on GitHub.

In summary, the immediate implications of our work are as follows:

- We discovered that existing literature does not provide a concrete landscape of human errors in software engineering. Our SLR (Section 4.1) revealed that 85% of existing studies do not have a scope general to all phases of software engineering, almost 50% of studies do not go beyond the basic human error types (*e.g.* slip, lapse, and mistake), and of those studies that do go beyond surface level categorization, 59% are at least

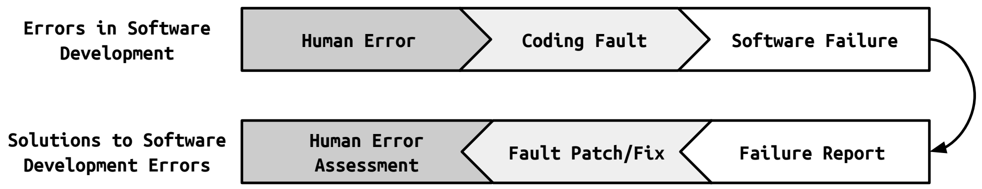


Figure 7.1: Complete Human Error Lifecycle in Software Engineering

Each stage of error in software development has its own solution. Software failures are addressed with failure reports and their underlying faults are patched/fixed. We demonstrate human error assessment (via micro post-mortems) as a process for addressing human errors in software engineering.

partially ambiguous. Further, our examination of software engineering artifacts (Section 4.2) identified 15 new categories of human error in software engineering that were not documented in existing literature.

- Our SLR (Section 4.1) and our user study (Section 5.2) revealed that the majority of software engineers' human errors are mistakes, a finding that is inconsistent with James Reason's findings [298]. Since mistakes are planning failures, and software engineering is a highly plan-oriented domain, this finding makes sense, but it also suggests that human error findings in other domains (*e.g.* aviation) may not generalize or apply to software engineering.
- Senior software engineering students believe human error assessment assisted by T.H.E.S.E. is a beneficial software engineering activity that leads to meaningful human error reflection, which would also be beneficial to professional software engineers.
- Our semi-automated refinement of T.H.E.S.E. categories (Section 5.3), revealed that classification of human errors using Sentence-BERT models may show promise.

In demonstrating the utility of T.H.E.S.E. and our micro post-mortem process, the software development community will be closer to inculcating the wisdom of historical developer human errors, enabling them to engineer higher quality and more secure software, closing the loop on the lifecycle of human error in software engineering (Figure 7.1). In the remainder of this section, we summarize the key contributions of this work, suggest recommendations for future work, and provide some closing thoughts and cautions.

7.1 Key Contributions

In addition to the findings discussed above, the key contributions of this dissertation are summarized as follows:

1. This dissertation establishes the **Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.)**, which contains 31 categories of human error spanning slips (8), lapses (8), and mistakes (15). T.H.E.S.E. is the culmination of nearly three years of research examining software engineers' human errors as documented in literature and development artifacts, and experienced during a user study. Version 4 of T.H.E.S.E. —which we refrain from calling the *final* version to leave room for and encourage future iterations—is included in Table 7.1. Extended descriptions for T.H.E.S.E. categories are provided in Appendix G and a full changelog for each version of T.H.E.S.E. is included in Appendix L.
2. This dissertation presents a **formal human error informed micro post-mortem process for software engineering** to accompany T.H.E.S.E. Our process has five steps—(1) summarize defect, (2) assign human error type, (3) assign T.H.E.S.E. category, (4) summarize human error, and (5) consider previous human errors—with guiding questions (outlined in Section 6.1). Our process is inspired by existing software engineering post-mortem processes [14, 40, 66, 81, 85, 162, 331, 359], human reliability analysis [4, 79, 134, 193, 335, 335], and after-event reviews [10, 86, 120]. Figure 6.1 visually summarizes our human error reflection process.
3. Finally, this dissertation includes the implementation of the **Human Error Reflection Engine (H.E.R.E.)**, a proof-of-concept workflow to facilitate human error reflection with T.H.E.S.E. on GitHub, lowering the barrier to entry for software engineers who wish to adopt human error reflection. H.E.R.E. is summarized in Section 6.2.

7.2 Recommendations for Future Work

While conducting this dissertation research, we encountered many interesting ideas and worthy questions that we deemed outside of our intended scope. In this section, we present some research directions as a starting point for researchers who wish to continue and build on this work.

Table 7.1: Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.) Version 4

ID	Source	Category/Definition
Slips		
S01	Artifacts (Section 4.2)	**Typos & Misspellings**: Typos and misspellings may occur in code comments, documentation (and other development artifacts), or when typing the name of a variable, function, or class. Examples include misspelling a variable name, writing down the wrong number/name/word during requirements elicitation, referencing the wrong function in a code comment, and inconsistent whitespace (that does not result in a syntax error).
S02	Literature (Section 4.1)	**Syntax Errors**: Any error in coding language syntax that impacts the executability of the code. Note that Logical Errors (*e.g.* `+=` instead of `+`) are not Syntax Errors. Examples include mixing tabs and spaces (*e.g.* Python), unmatched brackets/braces/-parenthesis/quotes, and missing semicolons (*e.g.* Java).
S03	Literature (Section 4.1)	**Overlooking Documented Information**: Errors resulting from overlooking (internally and externally) documented information, such as project descriptions, stakeholder requirements, API/library/tool/framework documentation, coding standards, programming language specifications, bug/issue reports, and looking at the wrong version of documentation or documentation for the wrong project/software.
S04	Artifacts (Section 4.2)	**Multitasking Errors**: Errors resulting from multitasking, *i.e.* working on multiple software engineering tasks at the same time.
S05	Artifacts (Section 4.2)	**Hardware Interaction Errors**: Attention failures while using computer peripherals, such as mice, keyboard, and cables. Examples include copy/paste errors, clicking the wrong button, using the wrong keyboard shortcut, and incorrectly plugging in cables.
S06	Artifacts (Section 4.2)	**Overlooking Proposed Code Changes**: Errors resulting from lack of attention during formal/informal code review. Examples include overlooking incorrect logic, or skipping files, functions, or classes during a review.
S07	User Study (Section 5.2)	**Overlooking Existing Functionality**: Errors resulting from overlooking existing functionality, such as reimplementing or duplicating variables, functions, and classes that already exist, or reimplementing functionality that already exists in a standard library. Other examples include deleting necessary variables, functions, and classes.
S08	—	**General Attentional Failure**: Only use this category if you believe your error to be the result of a lack of attention, but no other slip category fits.
Lapses		
L01	Artifacts (Section 4.2)	**Forgetting to Finish a Development Task**: Forgetting to finish a development task. Examples include forgetting to implement a required feature, forgetting to finish a user story, and forgetting to deploy a security patch.
L02	Literature (Section 4.1)	**Forgetting to Fix a Defect**: Forgetting to fix a defect that you encountered, but chose not to fix right away.
L03	Literature (Section 4.1)	**Forgetting to Remove Development Artifacts**: Forgetting to remove debug log files, dead code, informal test code, commented out code, test databases, backdoors, *etc.* Examples include leaving unnecessary code in the comments, and leaving notes in internal development documentation.
L04	Artifacts (Section 4.2)	**Working with Outdated Source Code**: Forgetting to git-pull (or equivalent in other version control systems), or using an outdated version of a library.
L05	Artifacts (Section 4.2)	**Forgetting an Import Statement**: Forgetting to import a necessary library, class, variable, or function, or forgetting to access a property, attribute, or argument. Examples include forgetting to import python's `sys` library, forgetting to include a header file in C, or forgetting to pass an argument to a function.
L06	Literature (Section 4.1)	**Forgetting to Save Work**: Forgetting to push code, or forgetting to backup/save data or documentation.
L07	Artifacts (Section 4.2)	**Forgetting Previous Development Discussion**: Errors resulting from forgetting details from previous development discussions.
L08	—	**General Memory Failure**: Only use this category if you believe your error to be the result of a memory failure, but no other lapse category fits.

continued on next page...

Table 7.1: Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.) Version 4 (Continued)

ID	Source	Category/Definition
Mistakes		
M01	Literature (Section 4.1)	**Code Logic Errors**: A code logic error is one in which the code executes (*i.e.* actually runs), but produces an incorrect output/behavior due to incorrect logic. Examples include using incorrect operators (*e.g.* `+=` instead of `+`), erroneous if/else statements, incorrect variable initializations, problems with variable scope, and omission of necessary logic.
M02	Literature (Section 4.1)	**Incomplete Domain Knowledge**: Errors resulting from incomplete knowledge of the software system's target domain (*e.g.* banking, astrophysics). Examples include planning/designing a system without understanding the nuances of the domain.
M03	Literature (Section 4.1)	**Wrong Assumption Errors**: Errors resulting from an incorrect assumption about system requirements, stakeholder expectations, project environments (*e.g.* coding languages and frameworks), library functionality, and program inputs.
M04	Literature (Section 4.1)	**Internal Communication Errors**: Errors resulting from inadequate communication between development team members. Examples include misunderstanding development discussion, misinterpretting or providing ambiguous instructions, communicating using the wrong medium (*e.g.* oral vs. written), or communicating ineffectively (*e.g.* too formal/informal, too much unnecessarily complex language, hostile language/body language).
M05	Literature (Section 4.1)	**External Communication Errors**: Errors resulting from inadequate communication with project stakeholders or third-party contractors. Examples include providing ambiguous or unclear directions to third-parties or users, or misinterpreting stakeholder feedback, communicating using the wrong medium (*e.g.* oral vs. written), or communicating ineffectively (*e.g.* too formal/informal, too much unecessarily complex language, hostile language/body language).
M06	Literature (Section 4.1)	**Solution Choice Errors**: Misunderstood problem-solving methods/techniques result in analyzing the problem incorrectly and choosing the wrong solution. For example, choosing to implement a database system in Python rather than using SQL, or choosing the wrong software design pattern. Overconfidence in a solution choice also falls under this category.
M07	Artifacts (Section 4.2)	**Time Management Errors**: Errors resulting from a lack of time management, such as failing to allocate enough time for the implementation of a feature, procrastinating a development task, or predicting the time required for a task incorrectly.
M08	Artifacts (Section 4.2)	**Inadequate Testing**: Failure to implement necessary test cases, failure to consider necessary test inputs, failure to implement a certain type of testing (*e.g.* unit, penetration, integration) when it is necessary, or failure to consider edge cases or unexpected inputs.
M09	Artifacts (Section 4.2)	**Incorrect/Insufficient Configuration**: Errors in configuration of libraries/frameworks/environments or errors related to missing configuration options. Examples include misconfigured IDEs or text editors, improper directory structure for a specific programming language, missing SSH keys, missing or incorrectly named database fields or tables, missing or incorrectly named/formatted configuration files, or not installing a required library.
M10	Literature (Section 4.1)	**Code Complexity Errors**: Errors resulting from misunderstood code due to poor documentation or unnecessary complexity. Examples include too many nested if/else statements or for-loops and poorly named variables/functions/classes/files.
M11	Artifacts (Section 4.2)	**Internationalization/String Encoding Errors**: Errors related to internationalization and/or string/character encoding. Examples include using ASCII instead of Unicode, using UTF8 when UTF16 was necessary, failure to design the system with internationalization in mind, or failing to verify the character length of user input.
M12	Artifacts (Section 4.2)	**Inadequate Experience Errors**: Errors resulting from inadequate experience with a language, library, framework, or tool.
M13	Artifacts (Section 4.2)	**Insufficient Tooling Access Errors**: Errors resulting from not having sufficient access to necessary tooling. Examples include not having access to a specific operating system, library, framework, hardware device, or not having the necessary permissions to complete a development task.
M14	Artifacts (Section 4.2)	**Workflow Order Errors**: Errors resulting from working out of order, such as implementing dependent features in the wrong order, implementing code before the design is stabilized, releasing code that is not ready to be released, or skipping a workflow step.
M15	—	**General Planning Failure**: Only use this category if you believe your error to be the result of a planning failure, but no other mistake category fits.

7.2.1 Human Errors in Other Software Engineering Artifacts

In Section 4.2 we examined software engineers' self-admitted human errors (identified using apology lemmas) in GitHub commits, issues, and pull-requests. While valuable, there is a wide range of other software engineering artifacts—*e.g.* security advisories, insider threat stories, developer conversations on platforms such as Bugzilla or JIRA—that should be examined for human errors. Also, as we noted in Section 4.2.4, plenty of self-admitted human errors exist beyond what is recorded in apologies, and plenty of human errors are not self-admitted. Those human errors, while outside of our scope, should also be studied to provide a fuller picture of human error in software engineering. Some potential research questions in this area include:

- How can software engineers' **non**-self-admitted human errors be identified?
- Which software engineering artifacts provide the most insight into human error?

7.2.2 Mitigation of Human Errors in Software Engineering

Peters & Peters [285] devote a chapter of their work to countermeasures (*i.e.* mitigations) of human error, proposing 12 basic principles for mitigating human error in order of most-to-least effective, which we quote below:

1. *Eliminate* the source of human error (remove the hazard). Make the error impossible by design.
2. *Control* the opportunity for error by physical means (engineering controls to prevent error, such as guards or barriers to prevent access to a source of error or hazard).
3. *Mitigate* the consequences of an error (risk severity reduction).
4. Ensure *detectability* of errors before damage occurs (foster immediate error correction).
5. Institute *procedural* pathways for guidance and to channel behavior (error avoidance restrictions and narrowing of conduct).
6. Maintain *supervisory* control and monitoring for errors (error observation, oral directions, and manual shutdown). Particularly useful for new tasks, new employees, new jobs, and new equipment.
7. Provide *instructions* that are written, brief, specific, and immediately available. However, unsupervised employees do not always follow instructions.
8. Utilize *training* to provide general background information, job context, knowledge about company culture, and safety rules. However, training may be remote in time and not specific to a job task.
9. Have *technical manuals* available for reference when questions arise or for general self-learning. They are useful to help avoid troubleshooting errors.
10. *Warnings* provide an informed opportunity to avoid harm. They are used for residual risks after the use of other remedies. Effective if well designed, read or heard, understood, and not disregarded.
11. Specify that *personal protective equipment* or other safety equipment be available when and where needed (for harm or injury reduction).
12. Assume intentional *risk acceptance* by no error prevention action and a toleration of the results. Provide appropriate insurance coverage or company reserves to compensate for the foreseeable harm. Maintain recall and public relations plans.

Peters & Peters [285] further suggest 26 specific countermeasures (see Appendix K). The countermeasure most relevant to this dissertation is *pragmatism*, the accurate and human-oriented categorization of human errors. These basic principles and specific countermeasures should be explored in the context of software engineering. Some suggested research questions in this area include:

- Which existing tools and processes in software engineering are effective mitigations for human error?
- What countermeasures are effective mitigations for each type of human error in software engineering?
- What countermeasures, if any, are applicable to multiple types of human error in software engineering?

7.2.3 Impact of Human Errors in Software Engineering

During our user study (Section 5.2), most participants indicated the belief that mistakes are more consequential than slips or lapses. For example:

> **Participant 3:** *"A majority of the slips and lapses affect just development and implementation, whereas a good chunk of mistakes can affect the entire process"*
>
> **Anonymous Feedback:** *"In my experience, the majority of slips and lapses can be fixed pretty easily, like a typo can be fixed in less than a second after you find it. Even if you misspelled it multiple times, your IDE will normally have like a refactor feature where you can change all references to a misspelled variable at once. Mistakes usually mean that you have to take extra time to fix something that went wrong. You don't really realize your mistake until it's too late and you're dealing with the consequences of that."*
>
> **Anonymous Feedback:** *"I would say mistakes have the most impact on a software project because slips and lapses have relatively simple solutions while mistakes lead to bigger changes in a software project."*

However, there is some disagreement:

> **Anonymous Feedback:** *"I think lapses are most likely to result from carelessness or a disregard for the project. These types of errors can result in the downfall of a project faster than the other two because the developers, who are supposed to be the most attentive to the project, aren't satisfying this requirement."*

Future work should explore the impact of slips versus lapses versus mistakes on software engineering projects. Knowing which type of human error impacts software engineering the most can help individual software engineers and their managers prioritize human error mitigations. Some potential research questions in this area include:

- What kinds of faults and failures typically result from slips, lapses, and mistakes?
- Which type of human error—slips, lapses, or mistakes—has the most impact (*i.e.* consequence) on a software project?

7.2.4 Symphonies of Human Errors in Software Engineering

In a study by Ko & Myers [168], developers were recorded thinking aloud in a series of experiments. Starting from an observed software error, runtime fault/failure, or cognitive breakdown, Ko & Myers identified the cause and documented chains of causes (*i.e.* chains of cognitive breakdowns) until no further cause could be identified. Ko & Myers observed 102 software errors stemming from 159 cognitive breakdowns. The average breakdown chain consisted of 2.3 breakdowns. This work is noteworthy for identifying that errors in software can be the result of multiple human errors at different stages of software development. During our user study, (Section 5.2), participants also described *symphonies* of human errors:

> **Participant 2:** *"Time management is the reason our stereotype is people coding for 24 hours straight, through the night, overloaded on caffeine... it's like ingrained in software development culture. Time management is just like totally, totally crucial and totally contributes to human errors a lot. It's like one of the biggest factors in my opinion."*
>
> **Participant 5:** *"I don't know what I did, but I completely broke debugging on my project. It was skipping over functions I was trying to step into and going to lines that weren't even code, just white space and comments. It might have been something misconfigured in my project, but everything I was doing to make it better only made it worse because I had no idea what I was doing."*

Some potential research questions in this area include:

- Are there overarching human error themes (*e.g.* time management) that lead to other human errors in software engineering?
- How often do symphonies of human errors occur in software engineering?
- Do certain kinds of software defects typically result from a symphony of human errors?
- Are there patterns of human error chains (*e.g.* `Internal Communication Errors (M04)` → `Solution Choice Errors (M06)`) in software engineering?

7.2.5 Inculcating a Human Error Mindset

Learning to think like an attacker (*i.e.* inculcating an attacker mindset) is an essential skill for software engineers, as an attacker mindset leads software engineers to ask questions such as *how can an attacker exploit this feature?* or *how can an attacker leverage this configuration to break into the system?*—this kind of speculative reasoning can help software engineers think about potential security flaws and work to prevent or mitigate vulnerabilities before they occur [204, 246, 247]. Future work should explore whether inculcating a *human error mindset* has similar benefits. During Phase 2 of our user study (Section 5.2), participants were asked if they found themselves thinking in terms of human error as their human errors were experienced, or if assessing human error was a more reflective process. At the beginning of Phase 2, participants responded with the following:

> **2022-09-16, Participant 2:** *"Rather than thinking about it in terms of... slip, lapse, mistake, I've been analyzing my errors closer, definitely because of this... with more focus on the reason itself than 'Oh, this is a mistake, or this was a lapse'... it's more like when I come across that mistake and when I eventually reach that solution, this was because I was rushing, or this is because I had unclear requirements, or a misunderstanding of my requirements. I think about the reason more closely now."*

> **2022-09-16, Participant 4:** *"More of a reflective process... because it was on a time crunch, I was more focused on like, just getting them fixed."*

> **2022-09-16, Participant 5:** *"Right now it's definitely more reflective, like I'm not seeing it in the moment."*

In the following weeks, some participants indicated a shift in their thought process, which indicates they may be internalizing human error:

> **2022-09-23, Participant 5:** *"I'm definitely keeping it in my mind more."*

> **2022-10-14, Participant 3:** *"When the [human error] actually happens, especially if it's a slip or lapse, it usually gets resolved in less than 5 minutes... although if it's a mistake, where there's like a long term consequence to it... I usually look back on that."*

> **2022-10-17, Participant 5:** *"A lot more. Like now, before I even fix it, when I realize something's wrong... I'll write it down and categorize it later."*

Future work should explore questions related to human error mindset, such as:

- How can software engineers inculcate a human error mindset?
- Does thinking about human errors in the moment, or before they occur, lead to improved software quality?
- What impact does inculcating a human error mindset have on individual software engineers' performance and on the performance of software engineering teams?

7.2.6 Reframing Historical Software Defects

Since every software defect (*i.e.* fault or failure) is the manifestation of human error [349], we encourage software engineering taxonomy researchers (notably those discussed in Section 3.3) to revisit their taxonomies and consider documented software engineering defects under the lens of human error.

For example, CWE collects the (mostly software-agnostic) underlying weaknesses that lead to a vulnerability. MITRE (the organization that maintains CWE) could re-examine existing CWE entries with human error in mind, similar to our approach in Section 5.3, and identify CWEs that are related to specific categories of human error. For example, `Code Logic Errors (M01)` encompasses many CWE entries, such as **CWE-478: Missing Default Case in Multiple Condition Expression** and **CWE-480: Use of Incorrect Operator**. CWE entries such as **CWE-477: Use of Obsolete Function** and **CWE-1068: Inconsistency Between Implementation and Documented Design** fall under `Overlooking Documented Information (S03)`.

What new insights can the software engineering community gain by examining historical defects and weaknesses with human error in mind? Some potential research questions in this area include:

- What types of human errors are documented in historical software defects?
- Are slips, lapses, and mistakes adequately represented in historical software defects?
- Are there new categories of human error documented in historical software defects?
- What relationships exist between human errors and vulnerabilities?

7.2.7 Automated Human Error Classification

Our classification experiment (Section 5.3) indicates that Sentence-BERT models may be able to accurately classify slips, lapses, and mistakes. However, we noted some limitations in this line of research that future researchers could address, including:

- Lack of training data: Our experiment used a small set of 368 manually labeled human errors in software engineering. We encourage future researchers to collect and annotate more data, and explore training models capable of classifying slips, lapses, and mistakes—both within software engineering and beyond. With more data, researchers may even be able to classify human errors beyond the surface level (*i.e.* classify T.H.E.S.E. categories).
- Lack of relevant models: With one exception [73], we could not find any Sentence-BERT models trained on software engineering data. We encourage future researchers to address this lack of software engineering specific language models. Researchers could also explore adapting other kinds of language models—such as Falcon [8], LLaMA [344], and GPT [291]—to human error classification and other tasks.

7.2.8 Socio-Linguistic Characteristics of Developers' Human Errors

In previous work, we examined socio-linguistic characteristics—*e.g.* politeness, uncertainty, syntactic complexity—of software engineers' conversations, revealing new insights into effective software engineering and security communication strategies [218, 219, 245]. Future work should examine similar socio-linguistic characteristics of software engineers' human errors. Some potential research questions in this area include:

- How does politeness impact the effectiveness of team-based human error reflection?
- How uncertain are software engineers when discussing their human errors?
- How much information is present in software engineers' human error discussions?
- Do software engineers with less experience and/or less project familiarity experience more human errors?

7.2.9 Human Error Prevention Through Existing Processes

In the final survey for our user study (Section 5.2), participants indicated that various tools and processes may be able to prevent or mitigate certain human errors. Participants indicated that code review, compilers, debuggers, continuous integration, and IDE tools could help identify and/or prevent slips. For lapses, participants suggested code review, regular note-taking, requirements testing, defect tracking, and maintaining sprint backlogs. Finally, for mistakes, participants suggested clearly defining and following processes, having a project manager be responsible for organization, regular verification with the project stakeholders, and quality assurance (*e.g.* checklists, control charts).

Some examples: if a software engineer uses an IDE, `Syntax Errors (S02)` can be mitigated because the syntax checker will identify syntax errors before they make it into the official source code; if a software engineering team tracks defects, `Forgetting to Fix a Defect (L02)` can be mitigated because software engineers can consult the defect log rather than trying to remember defects that need fixing; if a software engineering team performs regular code review, then many human errors (*e.g.* `Overlooking Documented Information (S03)` and `Code Logic Errors (M01)`) can be prevented because there are multiple software engineers examining the software product.

Researchers studying human error in software engineering should consider mapping existing software engineering processes to the human errors they intend to fix. This may even lead to human error informed risk assessment, *e.g.* identifying human errors that could be experienced during a project because certain tools/processes are not being used. Some potential research questions in this area include:

- Which software engineering processes are most effective at mitigating slips, lapses, and mistakes?
- Why do certain software engineering processes/tools identify or prevent human errors?
- Which human errors are not covered by existing software engineering processes?

7.2.10 Insights from Apologizing Software Engineers

Apology mining is a relatively new area of study, presenting many research opportunities. Future researchers should examine apologies in natural language across domains and consider the socio-technical factors that contribute to the culture of admitting to mistakes (in the general sense of the word). Some potential research questions in this area include:

- How frequently do practitioners apologize in domains such as medicine, transportation, first response, and politics?
- What factors contribute to the relative frequency of apologies in one domain versus others?
- How can a culture of admitting to mistakes be fostered in various domains?
- What value does admitting to mistakes provide?
- How can apologies be modeled in natural language?
- What socio-linguistic factors (*e.g.* politeness) are associated with apologies?
- Do open-source software engineers' admit to their human errors more frequently than closed-source developers?

7.3 Closing Thoughts & Cautions

Human error in software engineering is a growing, maturing domain with promising areas of study. However, there are some careful considerations to be made. We wish to encourage those reading this dissertation to view human error not as a *human* problem, but as an *organizational* problem. The old approach to human error was "beating the human until the error goes away [76]," but modern practitioners view human error as an opportunity to understand what went wrong and work toward prevention strategies [76]. Sydney Dekker, a human factors and safety expert, illustrates this point when discussing human errors in the aviation domain:

> *"Go back to the 1947 Fitts and Jones study. This is how their paper opened: "It should be possible to eliminate a large proportion of so-called 'pilot-error' accidents by designing equipment in accordance with human requirements [99]." 'Pilot error' was again put in quotations marks, and once it had been investigated properly, the solutions to it became rather obvious. The point was not the 'pilot error.' That was just the symptom of trouble, not the cause of trouble. It was just the starting point. The remedy did not lie in telling pilots not to make errors. Rather, Fitts and Jones argued, we should change the tools, fix the environment in which we make people work, and by that we can eliminate the errors of people who deal with those tools. Skill and experience, after all, had little influence on "error" rates: getting people trained better or disciplined better would not have much impact. Rather change the environment, and you change the behavior that goes on inside of it. Note also how Fitts and Jones did not call these episodes "failures." Instead, they used the neutral term "experiences." We could all learn a lot from their insights, their understanding, their open-mindedness and their moral maturity [76]."*

Throughout Section 4.2, we read many apologies made by software engineers, but this one effortlessly articulates the core of the human error mindset—a culture of acceptance, kindness, and progress in the face of human error:

> *"Apologies for the oversight... Unfortunately my mind is always split between the various Github, Forum, and Reddit issues (as well as the various features I'm working on), and that does sometimes lead to things slipping. I assure you it's not a result of malice, but just me being human. I encourage you to remind yourself of the human-ness of others when communicating online. I know this bug is frustrating, but hopefully this time around we'll get it for good :) [121]"*

Bibliography

[1] Robert P. Abbott, Janet S. Chin, James E. Donnelley, William L. Konigsford, S. Tokubo, and Douglas A. Webb. Security analysis and enhancements of computer operating systems. Technical report, Lawrence Livermore Laboratory, Institute for Computer Sciences and Technology/National Bureau of Standards, RISOS Project, Washington, DC, Apr 1976. Tech Report NBSIR 76-1041.

[2] Ritu Agarwal and Jayesh Prasad. A field study of the adoption of software process innovations by information systems professionals. *IEEE Transactions on Engineering Management*, 47(3):295–308, 2000.

[3] Munir Ahmed, Lukman Sharif, Muhammad Kabir, and Maha Al-Maimani. Human errors in information security. *International Journal*, 1(3):82–87, 2012.

[4] Tunc Aldemir. A survey of dynamic methodologies for probabilistic safety assessment of nuclear power plants. *Annals of Nuclear Energy*, 52:113–124, 2013.

[5] Ayman Alkhalifah, Alex Ng, ASM Kayes, Jabed Chowdhury, Mamoun Alazab, and Paul A. Watters. A taxonomy of blockchain threats and vulnerabilities. In *Blockchain for Cybersecurity and Privacy*, pages 3–28. CRC Press, 2020.

[6] Gary Allen and Roger Smith. After action review in military training simulations. In *Proceedings of Winter Simulation Conference*, pages 845–849. IEEE, 1994.

[7] Cecilia O. Alm, Benjamin S. Meyers, and Emily Prud'hommeaux. An analysis and visualization tool for case study learning of linguistic concepts. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 13–18, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[8] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. Falcon-40B: an open large language model with state-of-the-art performance. 2023.

[9] Edward G. Amoroso. *Fundamentals of computer security technology.* Prentice-Hall, Inc., 1994.

[10] Frederik Anseel, Filip Lievens, and Eveline Schollaert. Reflection as a strategy to enhance task performance after feedback. *Organizational Behavior and Human Decision Processes*, 110(1):23–35, 2009.

[11] Annie I. Antón and Julia B. Earp. A requirements taxonomy for reducing web site privacy vulnerabilities. *Requirements engineering*, 9(3):169–185, 2004.

[12] Vaibhav Anu, G.S. Walia, W. Hu, Jeffrey C. Carver, and Gary Bradshaw. Development of a taxonomy of requirements phase human errors using a systematic literature review: A technical report. *Technical Report*, 2016.

[13] Vaibhav Anu, Gursimran Walia, Wenhua Hu, Jeffrey C. Carver, and Gary Bradshaw. Error abstraction accuracy and fixation during error-based requirements inspections. In *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 45–46. IEEE, 2016.

[14] Vaibhav Anu, Gursimran Walia, Wenhua Hu, Jeffrey C. Carver, and Gary Bradshaw. The human error abstraction assist (HEAA) tool. Available online at `http://vaibhavanu.com/NDSU-CS-TP-2016-001.html`, 2016. Accessed 2021-10-28.

[15] Vaibhav Anu, Gursimran Walia, Wenhua Hu, Jeffrey C. Carver, and Gary Bradshaw. Using a cognitive psychology perspective on errors to improve requirements quality: An empirical investigation. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 65–76. IEEE, 2016.

[16] Vaibhav Anu, Gursimran Walia, and Gary Bradshaw. Incorporating human error education into software engineering courses via error-based inspections. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 39–44, 2017.

[17] Vaibhav Anu, Gursimran Walia, Wenhua Hu, Jeffrey C. Carver, and Gary Bradshaw. Issues and opportunities for human error-based requirements inspections: an exploratory study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 460–465. IEEE, 2017.

[18] Vaibhav Anu, Wenhua Hu, Jeffrey C. Carver, Gursimran S. Walia, and Gary Bradshaw. Development of a human error taxonomy for software requirements: a systematic literature review. *Information and Software Technology*, 103:112–124, 2018.

[19] Vaibhav Anu, Gursimran Walia, Gary Bradshaw, and Mohammad Alqudah. Developing and evaluating learning materials to introduce human error concepts in software engineering courses: Results from industry and academia. In *2019 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE, 2019.

[20] Vaibhav Anu, Kazi Zakia Sultana, and Bharath K. Samanthula. A human error based approach to understanding programmer-induced software vulnerabilities. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 49–54. IEEE, 2020.

[21] Vaibhav K. Anu, Gursimran S. Walia, Wenhua Hu, Jeffrey C. Carver, and Gary L. Bradshaw. Effectiveness of human error taxonomy during requirements inspection: An empirical investigation. In *SEKE*, pages 531–536, 2016.

[22] Vaibhav K. Anu, Gursimran Singh Walia, Gary L. Bradshaw, Wenhua Hu, and Jeffrey C. Carver. Using human error abstraction method for detecting and classifying requirements errors: A live study. In *REFSQ Workshops*. Citeseer, 2017.

[23] Vaibhav Kumar Anu. *Using Human Error Models to Improve the Quality of Software Requirements*. PhD thesis, North Dakota State University, 2018.

[24] Isaac Asimov. Liar!. astounding science fiction, 1941.

[25] Taimur Aslam. A taxonomy of security faults in the unix operating system. *Master's thesis, Purdue University*, 199(5), 1995.

[26] C. Richard Attanasio, Peter W. Markstein, and Ray J. Phillips. Penetrating an operating system: a study of vm/370 integrity. *IBM Systems Journal*, 15(1):102–116, 1976.

[27] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. Technical report, Citeseer, 2000.

[28] Alan David Baddeley. Working memory. *Philosophical Transactions of the Royal Society of London. B, Biological Sciences*, 302(1110):311–324, 1983.

[29] Victor R. Basili and Barry T. Perricone. Software errors and complexity: an empirical investigation. *Communications of the ACM*, 27(1):42–52, 1984.

[30] Victor R. Basili and H. Dieter Rombach. Tailoring the software process to project goals and environments. Technical report, Department of Computer Science, University of Maryland, 1986.

[31] Richard Baskerville. A taxonomy for analyzing hazards to information systems. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 167–176. Springer, 1996.

[32] Len Bass, Robert Nord, William Wood, and David Zubrow. Risk themes discovered through architecture evaluations. In *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, pages 1–1. IEEE, 2007.

[33] Himanshu Batra, Narinder Singh Punn, Sanjay Kumar Sonbhadra, and Sonali Agarwal. Bert-based sentiment analysis: A software engineering perspective. In *Database and Expert Systems Applications: 32nd International Conference, DEXA 2021, Virtual Event, September 27–30, 2021, Proceedings, Part I 32*, pages 138–148. Springer, 2021.

[34] Johannes Bauer and Christian Harteis. *Human fallibility: The ambiguity of errors for work and learning*, volume 6. Springer Science & Business Media, 2012.

[35] Paul Baybutt. Human factors in process safety and risk management: Needs for models, tools and techniques. In *Proceedings of the International Workshop on Human Factors in Offshore Operations, US Minerals Management Service, New Orleans*, pages 412–433, 1996.

[36] Kate Beeching. Apologies in french and english: An insight into conventionalisation and im/politeness. *Journal of Pragmatics*, 142:281–291, 2019.

[37] Boris Beizer. Software testing techniques, 1990.

[38] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* O'Reilly Media, Inc., 2009.

[39] Kamal Birdi, Janet Pennington, and Dieter Zapf. Ageing and errors in computer-based work: An observational field study. *Journal of Occupational and Organizational Psychology*, 70(1):35–47, 1997.

[40] Andreas Birk, Torgeir Dingsoyr, and Tor Stalhane. Postmortem: Never leave a project without it. *IEEE software*, 19(3):43–45, 2002.

[41] Richard Bisbey and Dennis Hollingworth. Protection analysis: Final report isi/sr-78-13. *USC/Information Sciences Institute*, 1978.

[42] Matt Bishop. A taxonomy of unix system and network vulnerabilities. Technical report, Citeseer, 1995.

[43] Matt Bishop. Vulnerabilities analysis. In *Proceedings of the Recent Advances in intrusion Detection*, pages 125–136, 1999.

[44] Matt Bishop. *How attackers break programs, and how to write programs more securely.* SANS Institute, 2002.

[45] Eeshita Biswas, Mehmet Efruz Karabulut, Lori Pollock, and K Vijay-Shanker. Achieving reliable sentiment analysis in the software engineering domain using bert. In *2020 IEEE International conference on software maintenance and evolution (ICSME)*, pages 162–173. IEEE, 2020.

[46] Harold Booth, Doug Rike, and Gregory Witte. The national vulnerability database (NVD): Overview, 12 2013. URL `https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=915172`.

[47] Ann Borkin and Susan M. Reinhart. Excuse me and i'm sorry. *TESOl Quarterly*, pages 57–69, 1978.

[48] Pierre Bourque and Richard E. Dupuis. Guide to the software engineering body of knowledge version 3.0 swebok. Available online at `https://www.computer.org/education/bodies-of-knowledge/software-engineering/v3`, 2014. Accessed 2021-10-13.

[49] John Bouvier. commission, n.d. In *A Law Dictionary, Adapted to the Constitution and Laws of the Unites States.* Farlex, Inc., 1856. Accessed 2021-11-09.

[50] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software*, 80(4):571–583, 2007.

[51] Aaron Brown and David A Patterson. Embracing failure: A case for recovery-oriented computing (roc). In *High Performance Transaction Processing Symposium*, volume 10, pages 3–8, 2001.

[52] Cati Brown, Tony Snodgrass, Susan J. Kemper, Ruth Herman, and Michael A. Covington. Automatic measurement of propositional idea density from part-of-speech tagging. *Behavior research methods*, 40(2):540–545, 2008.

[53] Penelope Brown, Stephen C Levinson, and Stephen C Levinson. *Politeness: Some universals in language usage*, volume 4. Cambridge university press, 1987.

[54] bry4n. About. `https://git-scm.com/about/branching-and-merging`, Jan 2014. Accessed 2021-09-23.

[55] bry4n, dbussink, peff, pepoirot, and schacon. git. `https://git-scm.com/`, Sep 2018. Accessed 2021-09-23.

[56] Michael D. Byrne and Susan Bovair. A working memory model of a common procedural error. *Cognitive science*, 21(1):31–61, 1997.

[57] Guillermo Campitelli and Fernand Gobet. Herbert simon's decision-making approach: Investigation of cognitive processes in experts. *Review of general psychology*, 14(4):354–364, 2010.

[58] James L. Cebula and Lisa R. Young. A taxonomy of operational cyber security risks. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 2010.

[59] ceggian. ceggian/sbert_pt_reddit_softmax_512. Available online at `https://huggingface.co/ceggian/sbert_pt_reddit_softmax_512`, 2022.

[60] Jonathan P. Chang, Caleb Chiam, Liye Fu, Andrew Wang, Justine Zhang, and Cristian Danescu-Niculescu-Mizil. Convokit: A toolkit for the analysis of conversations. In *Proceedings of the 21th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 57–60, 1st virtual meeting, July 2020. Association for Computational Linguistics.

[61] Nicholas Chantler. Profile of a computer hacker. *Faculty of Law, Queensland University of Technology, Australia*, 1996.

[62] Ram Chillarege, Inderpal S Bhandari, Jarir K Chaar, Michael J Halliday, Diane S Moebus, Bonnie K Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on software Engineering*, 18(11):943–956, 1992.

[63] Jim Christy. Cyber threat & legal issues. *Proceedings of the ShadowCon'99*, 1999.

[64] Trevor Cockram, Jim Salter, Keith Mitchell, Judith Cooper, Brian Kinch, and John May. Human error in the software generation process. In *Technology and Assessment of Safety-Critical Systems*, pages 175–185. Springer, 1994.

[65] Andrew D. Cohen and Elite Olshtain. Developing a measure of sociocultural competence: The case of apology. *Language learning*, 31(1):113–134, 1981.

[66] Chris Collison and Geoff Parcell. *Learning to Fly: Practical Lessons from one of the World's Leading Knowledge Companies.* Capstone, 2001.

[67] Mary Cummings. Informing autonomous system design through the lens of skill-, rule-, and knowledge-based behaviors. *Journal of Cognitive Engineering and Decision Making*, 12(1):58–61, 2018.

[68] Mary Missy Cummings. Man versus machine or man + machine? *IEEE Intelligent Systems*, 29(5):62–69, 2014.

[69] Michael A Cusumano and Richard W Selby. *Microsoft secrets: how the world's most powerful software company creates technology, shapes markets, and manages people.* Simon and Schuster, 1998.

[70] Cristian Danescu-Niculescu-Mizil, Moritz Sudhof, Dan Jurafsky, Jure Leskovec, and Christopher Potts. A computational approach to politeness with application to social factors. *arXiv preprint arXiv:1306.6078*, 2013.

[71] J. Dankelman, M. Wentink, C.A. Grimbergen, H.G. Stassen, and J. Reekers. Does virtual reality training make sense in interventional radiology? training skill-, rule-and knowledge-based behavior. *Cardiovascular and interventional radiology*, 27(5):417–421, 2004.

[72] Charles Darwin. *On the origin of species, 1859.* Routledge, 2004.

[73] Souvick Das, Novarun Deb, Agostino Cortesi, and Nabendu Chaki. Sentence embedding models for similarity detection of software requirements. *SN Computer Science*, 2:1–11, 2021.

[74] Fred D Davis, Richard P Bagozzi, and Paul R Warshaw. User acceptance of computer technology: A comparison of two theoretical models. *Management science*, 35(8):982–1003, 1989.

[75] Michael T DeGrosky. *Improving after action review (AAR) practice.* Citeseer, 2005.

[76] Sidney Dekker. From: The field guide to understanding human error. Presentation slides available online at `https://web.archive.org/web/20230720125059/https://static1.squarespace.com/static/53b78765e4b0949940758017/t/581c73ac29687fefa2d05bd2/1480886082793/Field+Guide+to+Understanding+Human+Error+Sidney-Dekker.pdf`, Feb 2013. Accessed 2021-10-14.

[77] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[78] Melvil Dewey. *A classification and subject index, for cataloguing and arranging the books and pamphlets of a library.* Brick row book shop, Incorporated, 1876.

[79] Valentina Di Pasquale, Raffaele Iannone, Salvatore Miranda, Stefano Riemma, et al. An overview of human reliability analysis techniques in manufacturing operations. *Operations management*, 9:978–953, 2013.

[80] Torgeir Dingsøyr. Postmortem reviews: purpose and approaches in software engineering. *Information and Software Technology*, 47(5):293–303, 2005.

[81] Torgeir Dingsøyr, Nils Brede Moe, and Ø; ystein Nytrø. Augmenting experience reports with lightweight postmortem reviews. In *International Conference on Product Focused Software Process Improvement*, pages 167–181. Springer, 2001.

[82] Docker Inc. Use containers to Build, Share and Run your applications. Available online at `https://www.docker.com/resources/what-container/`, 2023. Accessed 2023-08-25.

[83] Molla S Donaldson, Janet M Corrigan, Linda T Kohn, et al. To err is human: building a safer health system. *Committee on Quality of Health Care in America*, 2000.

[84] Wenliang Du and Aditya P. Mathur. Categorization of software errors that led to security breaches. In *21st National Information Systems Security Conference*, pages 392–407, 1998.

[85] Tore Dybå, Torgeir Dingsøyr, and Nils Brede Moe. *Process Improvement in Practice: A Handbook for IT Companies*, volume 9. Springer Science & Business Media, 2004.

[86] Shmuel Ellis. Learning from errors: The role of after-event reviews. In *Human fallibility: The ambiguity of errors for work and learning*, pages 215–232. Springer, 2012.

[87] David Embrey. Understanding human behaviour and error. *Human Reliability Associates*, 1(2005):1–10, 2005.

[88] espejelomar and nreimers. all-mpnet-base-v2. Available online at `https://huggingface.co/sentence-transformers/all-mpnet-base-v2`, 2022.

[89] espejelomar and nreimers. multi-qa-distilbert-cos-v1. Available online at `https://huggingface.co/sentence-transformers/multi-qa-distilbert-cos-v1`, 2022.

[90] espejelomar and nreimers. multi-qa-mpnet-base-dot-v1. Available online at `https://huggingface.co/sentence-transformers/multi-qa-mpnet-base-dot-v1`, 2022.

[91] espejelomar, nreimers, and guillaume-be. all-MiniLM-L12-v2. Available online at `https://huggingface.co/sentence-transformers/all-MiniLM-L12-v2`, 2022.

[92] espejelomar, nreimers, and guillaume-be. all-distilroberta-v1. Available online at `https://huggingface.co/sentence-transformers/all-distilroberta-v1`, 2022.

[93] espejelomar, nreimers, and joaogante. multi-qa-MiniLM-L6-cos-v1. Available online at `https://huggingface.co/sentence-transformers/multi-qa-MiniLM-L6-cos-v1`, 2022.

[94] espejelomar, nreimers, and joaogante. paraphrase-MiniLM-L3-v2. Available online at `https://huggingface.co/sentence-transformers/paraphrase-MiniLM-L3-v2`, 2022.

[95] espejelomar, nreimers, joaogante, and guillaume-be. paraphrase-albert-small-v2. Available online at `https://huggingface.co/sentence-transformers/paraphrase-albert-small-v2`, 2022.

[96] espejelomar, nreimers, joaogante, guillaume-be, and pfr. all-MiniLM-L6-v2. Available online at `https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2`, 2022.

[97] Richárd Farkas, Veronika Vincze, György Móra, János Csirik, and György Szarvas. The conll-2010 shared task: learning to detect hedges and their scope in natural language text. In *Proceedings of the fourteenth conference on computational natural language learning–Shared task*, pages 1–12, 2010.

[98] Robin E. Ferner and Jeffrey K. Aronson. Clarification of terminology in medication errors. *Drug safety*, 29(11): 1011–1022, 2006.

[99] P.M. Fitts and R.E. Jones. *Analysis of factors contributing to 460 "pilot-error" experiences in operating aircraft controls.* Dayton, OH: Aero MEdical Laboratory, Air Material Command, Wright-Patterson Air Force Base, 1947.

[100] Rudolph Flesch. A new readability yardstick. *Journal of applied psychology*, 32(3):221, 1948.

[101] Forum of Incident Response and Security Teams (FIRST). Common Vulnerability Scoring System version 3.1: Specification Document. Available online at `https://www.first.org/cvss/specification-document`, 2023. Accessed 2023-08-03.

[102] Lyn Frazier. Syntactic complexity. *Natural language parsing: Psychological, computational, and theoretical perspectives*, pages 129–189, 1985.

[103] Lyn Frazier. Sentence processing: A tutorial review. *Attention and performance XII: the psychology of reading*, 1987.

[104] Lyn Frazier, Lori Taft, Tom Roeper, Charles Clifton, and Kate Ehrlich. Parallel structure: A source of facilitation in sentence comprehension. *Memory & cognition*, 12(5):421–430, 1984.

[105] Sigmund Freud. *The Psychopathology of Everyday Life.* Read Books Limited, 2014. ISBN 9781473396234. Originally published 1901.

[106] Zachary P. Fry and Westley Weimer. A human study of fault localization accuracy. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[107] David A Garvin. *Learning in action: A guide to putting the learning organization to work.* Harvard Business Review Press, 2003.

[108] Gerd Gigerenzer and Reinhard Selten. *Bounded rationality: The adaptive toolbox.* MIT press, 2002.

[109] GitHub. Advanced search. Available online at `https://github.com/search/advanced`, Aug 2021. Accessed 2021-08-31.

[110] GitHub. Where the world builds software. `https://github.com/about`, 2021. Accessed 2021-09-23.

[111] GitHub. Who's using github? `https://government.github.com/community/`, 2021. Accessed 2021-09-23.

[112] GitHub. GraphQL API. Available online at `https://docs.github.com/en/graphql`, 2022. Accessed 2022-04-28.

[113] GitHub. About pull request reviews. Available online at `https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/reviewing-changes-in-pull-requests/about-pull-request-reviews`, Feb 2022. Accessed 2022-05-01.

[114] GitHub. Github actions. Available online at `https://github.com/features/actions`, 2023. Accessed 2023-07-06.

[115] Robert L. Glass. Persistent software errors. *IEEE transactions on Software Engineering*, SE-7(2):162–168, 1981.

[116] J.J. Godfrey, E.C. Holliman, and J. McDaniel. SWITCHBOARD: Telephone speech corpus for research and development. In *Proceedings of the 1992 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-92)*, volume 1, pages 517–520, 1992. doi: 10.1109/ICASSP.1992.225858.

[117] Peter Godfrey-Smith. *Other minds: The octopus, the sea, and the deep origins of consciousness.* Farrar, Straus and Giroux, 2016.

[118] R Graham. Bikers learn from the army: Harley–davidson uses afteraction reviews to build a smarter production process. *Knowledge Management*, 2001.

[119] Andy Gray. An historical perspective of software vulnerability management. *Information Security Technical Report*, 8(4):34–44, 2003.

[120] David E Gray. Facilitating management learning: Developing critical reflection through reflective tools. *Management learning*, 38(5):495–517, 2007.

[121] greyson-signal. MMS Receiving not working with non signal users. Available online at `https://github.com/signalapp/Signal-Android/issues/8571#issuecomment-544353977`, 10 2019. Accessed 2023-07-07.

[122] Herbert P. Grice. Logic and conversation. In *Speech acts*, pages 41–58. Brill, 1975.

[123] Leslie N Gruis. The mathematics of tic-tac-toe. Available online at `https://www.crows.org/blogpost/1685693/357431/The-Mathematics-of-Tic-Tac-Toe`, 2020.

[124] Dina Guglielmi, Alessio Paolucci, Valerio Cozzani, Marco Giovanni Mariani, Luca Pietrantoni, and Federico Fraboni. Integrating human barriers in human reliability analysis: A new model for the energy sector. *International Journal of Environmental Research and Public Health*, 19(5):2797, 2022.

[125] Robert Hackett. Stagefright: Everything you need to know about google's android megabug. Available online at `https://fortune.com/2015/07/28/stagefright-google-android-security/`, July 2015. Accessed 2023-08-01.

[126] Sara Hajian, Faramarz Hendessi, and Mehdi Berenjkoub. A taxonomy for network vulnerabilities. *International Journal of Information and Communication Technology Research*, 2010.

[127] Simon Hansman and Ray Hunt. A taxonomy of network and computer attacks. *Computers & Security*, 24(1): 31–43, 2005.

[128] Francis Heylighen and Jean-Marc Dewaele. Formality of language: definition, measurement and behavioral determinants. *Interner Bericht, Center "Leo Apostel", Vrije Universiteit Brüssel*, 4, 1999.

[129] Gerald N. Hill and Kathleen T. Hill. omission, n.d. In *The People's Law Dictionary*. Farlex, Inc., 1981-2005. Accessed 2021-11-09.

[130] Alan Hobbs and Ann Williamson. Skills, rules and knowledge in aircraft maintenance: errors in context. *Ergonomics*, 45(4):290–308, 2002.

[131] Richard C. Hollinger. Evidence that computer crime follows a guttman-like progression. *Sociology and Social Research*, 72(3):199–200, 1988.

[132] Erik Hollnagel. Human reliability analysis. *Context and control*, 1993.

[133] Janet Holmes. Apologies in new zealand english. *Language in society*, 19(2):155–199, 1990.

[134] Lin-Xiu Hou, Ran Liu, Hu-Chen Liu, and Shan Jiang. Two decades on human reliability analysis: a bibliometric analysis and literature review. *Annals of Nuclear Energy*, 151:107969, 2021.

[135] John D. Howard and Thomas A. Longstaff. A common language for computer security incidents. Technical report, Sandia National Labs., Albuquerque, NM (US); Sandia National Labs., Livermore, CA (US), 1998.

[136] John Douglas Howard. *An analysis of security incidents on the internet 1989-1995*. Carnegie Mellon University, 1997.

[137] Michael Howard, David LeBlanc, and John Viega. *24 deadly sins of software security: Programming flaws and how to fix them*. McGraw-Hill Education, 2010. Availabile online at `https://ebookcentral.proquest.com/lib/rit/detail.action?docID=4657697`.

[138] Wenhua Hu. *Application of human error theories in detecting and preventing software requirement errors*. The University of Alabama, 2017.

[139] Wenhua Hu, Jeffrey C. Carver, Vaibhav K. Anu, Gursimran S. Walia, and Gary Bradshaw. Detection of requirement errors and faults via a human error taxonomy: a feasibility study. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2016.

[140] Wenhua Hu, Jeffrey C. Carver, Vaibhav Anu, Gursimran Walia, and Gary Bradshaw. Defect prevention in requirements using human error information: An empirical study. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 61–76. Springer, 2017.

[141] Wenhua Hu, Jeffrey C. Carver, Vaibhav Anu, Gursimran S. Walia, and Gary L. Bradshaw. Using human error information for error prevention. *Empirical Software Engineering*, 23(6):3768–3800, 2018.

[142] Fuqun Huang. Post-completion error in software development. In *2016 IEEE/ACM Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 108–113. IEEE, 2016.

[143] Fuqun Huang, Bin Liu, and Bing Huang. A taxonomy system to identify human error causes for software defects. In *The 18th international conference on reliability and quality in design*, pages 44–49, 2012.

[144] Fuqun Huang, Bin Liu, You Song, and Shreya Keyal. The links between human error diversity and software diversity: Implications for fault diversity seeking. *Science of Computer Programming*, 89:350–373, 2014.

[145] IEEE. Ieee standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, 2010. doi: 10.1109/IEEESTD.2010.5399061.

[146] Vinay M. Igure and Ronald D. Williams. Taxonomies of attacks and vulnerabilities in computer systems. *IEEE Communications Surveys & Tutorials*, 10(1):6–19, 2008.

[147] Ghi Paul Im and Richard L. Baskerville. A longitudinal study of information system threat categories: the enduring problem of human error. *ACM SIGMIS Database: the DATABASE for Advances in Information Systems*, 36(4):68–79, 2005.

[148] Juha Itkonen, Mika V. Mäntylä, and Casper Lassenius. The role of the tester's knowledge in exploratory software testing. *IEEE Transactions on Software Engineering*, 39(5):707–724, 2012.

[149] N.D. Jayaram and P.L.R. Morse. Network security-a taxonomic view. In *European Conference on Security and Detection, 1997. ECOS 97.*, pages 124–127. IET, 1997.

[150] jhgan and joaogante. jhgan/ko-sbert-sts. Available online at `https://huggingface.co/jhgan/ko-sbert-sts`, 2022.

[151] jhgan and joaogante. jhgan/ko-sroberta-multitask. Available online at `https://huggingface.co/jhgan/ko-sroberta-multitask`, 2022.

[152] Kanta Jiwnani and Marvin Zelkowitz. Maintaining software with a security perspective. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 194–203. IEEE, 2002.

[153] Kanta Jiwnani and Marvin Zelkowitz. Susceptibility matrix: A new aid to software auditing. *IEEE security & privacy*, 2(2):16–21, 2004.

[154] Nina Viktoria Juliadotter and Kim-Kwang Raymond Choo. Cloud attack and risk assessment taxonomy. *IEEE Cloud Computing*, 2(1):14–20, 2015.

[155] julien-c, nreimers, joaogante, patrickvonplaten, and osanseviero. LaBSE. Available online at `https://huggingface.co/sentence-transformers/LaBSE`, 2022.

[156] Daniel Jurafsky, Elizabeth Shriberg, and Debra Biasca. Switchboard SWBD-DAMSL shallow-discourse-function annotation coders manual, draft 13. Technical Report 97-02, University of Colorado, Boulder Institute of Cognitive Science, Boulder, CO, 1997.

[157] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[158] SG Kariuki and K Löwe. Integrating human factors into process hazard analysis. *Reliability Engineering & System Safety*, 92(12):1764–1773, 2007.

[159] Richard N. Keers, Steven D. Williams, Jonathan Cooke, and Darren M. Ashcroft. Causes of medication administration errors in hospitals: a systematic review of quantitative and qualitative evidence. *Drug safety*, 36(11):1045–1067, 2013.

[160] Brandon Keller, Andrew Meneely, and Benjamin Meyers. What happens when we fuzz? investigating oss-fuzz bug history. In *International Conference on Mining Software Repositories (MSR)*, 2023.

[161] Lorenzo Keller, Prasang Upadhyaya, and George Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 157–166. IEEE, 2008.

[162] Norman Kerth. *Project retrospectives: a handbook for team reviews.* Addison-Wesley, 2013.

[163] Kevin S. Killourhy, Roy A. Maxion, and Kymie M.C. Tan. A defense-centric taxonomy based on attack manifestations. In *International Conference on Dependable Systems and Networks, 2004*, pages 102–111. IEEE, 2004.

[164] Mijung Kim, Saurabh Sinha, Carsten Görg, Hina Shah, Mary Jean Harrold, and Mangala Gowri Nanda. Automated bug neighborhood analysis for identifying incomplete bug fixes. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 383–392. IEEE, 2010.

[165] Tereza G. Kirner and Janaina C. Abib. Inspection of software requirements specification documents: A pilot study. In *Proceedings of the 15th annual international conference on Computer documentation*, pages 161–171, 1997.

[166] Eric Knight. Computer vulnerabilities. *Retrieved March*, 2000.

[167] Donald E. Knuth. The errors of TeX. *Software: Practice and Experience*, 19(7):607–685, 1989.

[168] Andrew J. Ko and Brad A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1-2):41–84, 2005.

[169] Jeno Kollarits. Über lagewahrnehmungsfehler und über einige andere besonderheiten des hypnopompiums. *Deutsche Zeitschrift für Nervenheilkunde*, 144(5):277–289, 1937.

[170] kowshikBlue. BlueAvenir/sti_security_class_model. Available online at `https://huggingface.co/BlueAvenir/sti_security_class_model`, 2023.

[171] Sara Kraemer and Pascale Carayon. Human errors and violations in computer and information security: The viewpoint of network administrators and security specialists. *Applied ergonomics*, 38(2):143–154, 2007.

[172] Daniela Kramer-Moore and Michael Moore. Pardon me for breathing: Seven types of apology. *ETC: A review of general semantics*, 60(2):160–169, 2003.

[173] Ivan Krsul. Computer vulnerability analysis: Thesis proposal. *Department of Computer Science, Purdue University*, 1997.

[174] Ivan Victor Krsul. *Software vulnerability analysis.* Purdue University, 1998.

[175] kwoncho. kwoncho/ko-sroberta-multitask-suspicious. Available online at `https://huggingface.co/kwoncho/ko-sroberta-multitask-suspicious`, 2023.

[176] Shibamouli Lahiri. Squinky! a corpus of sentence-level formality, informativeness, and implicature. *arXiv preprint arXiv:1506.02306*, 2015.

[177] Jung-Ying Lai, Jain-Shing Wu, Shih-Jen Chen, Chia-Huan Wu, and Chung-Huang Yang. Designing a taxonomy of web attacks. In *2008 International Conference on Convergence and Hybrid Information Technology*, pages 278–282. IEEE, 2008.

[178] Robin Tolmach Lakoff. Nine ways of looking at apologies: The necessity for interdisciplinary theory and method in discourse analysis. *The handbook of discourse analysis*, pages 199–214, 2001.

[179] Bill Landreth and Howard Rheingold. *Out of the inner circle: a hacker's guide to computer security.* Microsoft Press Bellevue, Washington, 1985.

[180] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys (CSUR)*, 26(3):211–254, 1994.

[181] Filippo Lanubile, Forrest Shull, and Victor R Basili. Experimenting with error abstraction in requirements documents. In *Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No. 98TB100262)*, pages 114–121. IEEE, 1998.

[182] Lucian L. Leape. Error in medicine. *Jama*, 272(23):1851–1857, 1994.

[183] Marek Leszak, Dewayne E Perry, and Dieter Stoll. A case study in root cause defect analysis. In *Proceedings of the 22nd international conference on Software engineering*, pages 428–437, 2000.

[184] Lingjia Li, Jian Cao, and Qin Qi. Monitoring negative sentiment-related events in open source software projects. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, pages 92–100. IEEE, 2021.

[185] Chiuhsiang Joe Lin, Tzu-Chung Yenn, and Chih-Wei Yang. Optimizing human–system interface automation design based on a skill-rule-knowledge framework. *Nuclear Engineering and Design*, 240(7):1897–1905, 2010.

[186] Chiuhsiang Joe Lin, Wei-Jung Shiang, Chun-Yu Chuang, and Jin-Liang Liou. Applying the skill-rule-knowledge framework to understanding operators' behaviors and workload in advanced main control rooms. *Nuclear Engineering and Design*, 270:176–184, 2014.

[187] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jinsong Dong. Break the dead end of dynamic slicing: Localizing data and control omission bug. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 509–519, 2018.

[188] Ulf Lindqvist and Erland Jonsson. How to systematically classify computer security intrusions. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)*, pages 154–163. IEEE, 1997.

[189] Richard Lippmann, Joshua W Haines, David J. Fried, Jonathan Korba, and Kumar Das. The 1999 darpa off-line intrusion detection evaluation. *Computer networks*, 34(4):579–595, 2000.

[190] Richard P. Lippmann, David J. Fried, Isaac Graf, Joshua W. Haines, Kristopher R. Kendall, David McClung, Dan Weber, Seth E. Webster, Dan Wyschogrod, Robert K. Cunningham, and Marc A. Zissman. Evaluating intrusion detection systems: The 1998 darpa off-line intrusion detection evaluation. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 12–26. IEEE, 2000.

[191] Tamara Lopez, Marian Petrel, and Bashar Nuseibehl. Examining active error in software development. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 152–156. IEEE, 2016.

[192] Daniel Lowry Lough. *A taxonomy of computer attacks with applications to wireless networks.* Virginia Polytechnic Institute and State University, 2001.

[193] Melinda Lyons, Sally Adams, Maria Woloshynowych, and Charles Vincent. Human reliability analysis in healthcare: a review of techniques. *International Journal of Risk & Safety in Medicine*, 16(4):223–237, 2004.

[194] Kavitha Manjunath, Vaibhav Anu, Gursimran Walia, and Gary Bradshaw. Training industry practitioners to investigate the human error causes of requirements faults. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 53–58. IEEE, 2018.

[195] David E. Mann and Steven M. Christey. Towards a common enumeration of vulnerabilities. In *2nd Workshop on Research with Security Vulnerability Databases, Purdue University, West Lafayette, Indiana*, 1999.

[196] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *An Introduction to Information Retrieval.* Cambridge University Press, 2008. ISBN 9780521865715.

[197] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[198] Timo Mantere and Jarmo T Alander. Evolutionary software engineering, a review. *Applied Soft Computing*, 5 (3):315–331, 2005.

[199] Solomon Marcus. Mathematical and computational mistakes and failures as a source of creativity. In *2010 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 8–11. IEEE, 2010.

[200] Brian Marick. A survey of software fault surveys. Technical Report 1651, Department of Computer Science. University of Illinois at Urbana-Champaign, 1990.

[201] Bob Martin and Steve Christey Coley. Common Weakness Scoring System (CWSS). `https://cwe.mitre.org/cwss/cwss_v1.0.1.html`, 2014. Accessed 2020-09-01.

[202] Robert G. Mays, Carole L. Jones, Gerald J. Holloway, and Donald P. Studinski. Experiences with defect prevention. *IBM Systems Journal*, 29(1):4–32, 1990.

[203] Sarah E. McDowell, Harriet S. Ferner, and Robin E. Ferner. The pathophysiology of medication errors: how and where they arise. *British journal of clinical pharmacology*, 67(6):605–613, 2009.

[204] Gary McGraw. *Software Security: Building Security in.* Addison-Wesley professional computing series. Addison-Wesley, 2006. ISBN 9780321356703.

[205] William S. McPhee. Operating system integrity in os/vs2. *IBM Systems Journal*, 13(3):230–252, 1974.

[206] Ardith J. Meier. Apologies: What do we know? *International journal of applied linguistics*, 8(2):215–231, 1998.

[207] Peter Mell, Karen Scarfone, and Sasha Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6):85–89, 2006.

[208] Andrew Meneely. What is secure? Lecture for Engineering Secure Software course, `https://web.archive.org/web/20230720130433/https://www.se.rit.edu/~se331/lectures/1-1%20Introduction.pptx`, 2012.

[209] Andy Meneely, Derek Leung, Marianna Sternefeld, Eli Bosley, Catherine Osadciw, Grant Grubbs, Bryan Quinn, kayladavis, Alex McHugh, Bryce Murphy, jlt8213, Kayla Nussbaum, Hari Dahal, roshnib212, Stephen Cioffi, Elijah Cantella, Lipee Vora, Dylan Green, Michael Padovani, Ryan Borger, and Matthew Schmitt. VulnerabilityHistoryProject/vulnerability-history: Version 1.0!, Oct 2020. URL `https://doi.org/10.5281/zenodo.4161682`.

[210] Guozhu Meng, Yang Liu, Jie Zhang, Alexander Pokluda, and Raouf Boutaba. Collaborative security: A survey and taxonomy. *ACM Computing Surveys (CSUR)*, 48(1):1–42, 2015.

[211] Merriam-Webster.com. apology, noun. In *Merriam-Webster.com*. Merriam-Webster.com, Apr 2022. Accessed 2022-04-30.

[212] Benjamin S Meyers. meyersbs/developer-apologies. Available online at `https://github.com/meyersbs/developer-apologies`, 2022. Accessed 2022-06-10.

[213] Benjamin S. Meyers and Andrew Meneely. An automated post-mortem analysis of vulnerability relationships using natural language word embeddings. *Procedia Computer Science*, 184:953–958, 2021.

[214] Benjamin S. Meyers and Andrew Meneely. 88.6 million developer comments from github. Available online at `https://doi.org/10.5281/zenodo.5603093`, Oct 2021.

[215] Benjamin S. Meyers and Andrew Meneely. 1,237 annotated developer apologies from github. Available online at `https://doi.org/10.5281/zenodo.10079441`, Nov 2023.

[216] Benjamin S. Meyers and Andrew Meneely. 200 annotated developer human errors from github. Available online at `https://doi.org/10.5281/zenodo.10080449`, Nov 2023.

[217] Benjamin S. Meyers and Andrew Meneely. 162 human error descriptions and categorizations from a user study. Available online at `https://doi.org/10.5281/zenodo.10079277`, Nov 2023.

[218] Benjamin S. Meyers, Nuthan Munaiah, Emily Prud'hommeaux, Andrew Meneely, Cecilia O. Alm, Josephine Wolff, and Pradeep K. Murukannaiah. A dataset for identifying actionable feedback in collaborative software development. In *Proceedings of the 2018 Meeting for the Association for Computational Linguistics*, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[219] Benjamin S. Meyers, Nuthan Munaiah, Andrew Meneely, and Emily Prud'hommeaux. Pragmatic characteristics of security conversations: An exploratory linguistic analysis. In *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, Montreal, QC, Canada, May 2019. IEEE/ACM.

[220] Benjamin S Meyers, Sultan Fahad Almassari, Brandon N Keller, and Andrew Meneely. Examining penetration tester behavior in the collegiate penetration testing competition. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–25, 2022.

[221] C.A. Meyers, S.S. Powers, and D.M. Faissol. Taxonomies of cyber adversaries and attacks: a survey of incidents and approaches. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2009.

[222] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[223] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[224] George A. Miller, Galanter Eugene, and Karl H. Pribram. *Plans and the Structure of Behaviour*. Routledge, 1960.

[225] MITRE. CVE-2014-0160. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160`, 2014. Accessed 2023-07-28.

[226] MITRE. CVE-2015-1538. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-1538`, 2015. Accessed 2023-08-01.

[227] MITRE. CVE-2015-1539. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-1539`, 2015. Accessed 2023-08-01.

[228] MITRE. CVE-2015-3824. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-3824`, 2015. Accessed 2023-08-01.

[229] MITRE. CVE-2015-3826. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-3826`, 2015. Accessed 2023-08-01.

[230] MITRE. CVE-2015-3827. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-3827`, 2015. Accessed 2023-08-01.

[231] MITRE. CVE-2015-3828. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-3828`, 2015. Accessed 2023-08-01.

[232] MITRE. CVE-2015-3829. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-3829`, 2015. Accessed 2023-08-01.

[233] MITRE. Common attack pattern enumeration and classification (CAPEC). `https://capec.mitre.org/about/index.html`, Apr 2019. Accessed 2021-09-09.

[234] MITRE. CVE-2020-9371. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-9371`, 2020. Accessed 2021-09-09.

[235] MITRE. Common weakness enumeration (CWE). `https://cwe.mitre.org/about/index.html`, Aug 2020. Accessed 2021-09-09.

[236] MITRE. About CWE. `https://cwe.mitre.org/about/history.html`, Feb 2020. Accessed 2021-09-14.

[237] MITRE. Common vulnerabilities and exposures (CVE). `https://cve.mitre.org/`, Sep 2021. Accessed 2021-09-09.

[238] MITRE. CWE-707: Improper Neutralization. `https://cwe.mitre.org/data/definitions/707.html`, Jul 2021. Accessed 2021-09-09.

[239] MITRE. CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection'). `https://cwe.mitre.org/data/definitions/74.html`, Jul 2021. Accessed 2021-09-09.

[240] MITRE. CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'). `https://cwe.mitre.org/data/definitions/79.html`, Jul 2021. Accessed 2021-09-09.

[241] MITRE. D3FEND, a knowledge graph of cybersecurity countermeasures. `https://d3fend.mitre.org/`, 2021. Accessed 2021-09-14.

[242] Akira Miyake and Priti Shah. *Models of working memory: Mechanisms of active maintenance and executive control*. Cambridge University Press, 1999.

[243] Ines Montani, Matthew Honnibal, Matthew Honnibal, Sofie Van Landeghem, Adriane Boyd, Henning Peters, Paul O'Leary McCann, Maxim Samsonov, Jim Geovedi, Jim O'Regan, Duygu Altinok, György Orosz, Søren Lind Kristiansen, Roman, Explosion Bot, Lj Miranda, Leander Fiedler, Daniël de Kok, Grégory Howard, Edward, Wannaphong Phatthiyaphaibun, Yohei Tamura, Sam Bozek, murat, Mark Amery, Ryn Daniels, Björn Böing, Pradeep Kumar Tippa, and Peter Baumgartner. explosion/spaCy: v3.1.6: Workaround for Click/Typer issues, 3 2022. URL `https://doi.org/10.5281/zenodo.6397450`.

[244] Sara Mumtaz, Carlos Rodriguez, Boualem Benatallah, Mortada Al-Banna, and Shayan Zamanirad. Learning word representation for the cyber security vulnerability domain. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

[245] Nuthan Munaiah, Benjamin S. Meyers, Cecilia O. Alm, Andrew Meneely, Pradeep K. Murukannaiah, Emily Prud'hommeaux, Josephine Wolff, and Yang Yu. Natural language insights from code reviews that missed a vulnerability. In *International Symposium on Engineering Secure Software and Systems*, pages 70–86, Bonn, Germany, August 2017. Springer.

[246] Nuthan Munaiah, Justin Pelletier, Shau-Hsuan Su, S Jay Yang, and Andrew Meneely. A cybersecurity dataset derived from the national collegiate penetration testing competition. In *HICSS Symposium on cybersecurity big data analytics*, 2019.

[247] Nuthan Munaiah, Akond Rahman, Justin Pelletier, Laurie Williams, and Andrew Meneely. Characterizing attacker behavior in a cybersecurity penetration testing competition. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6. IEEE, 2019.

[248] Tewodros Legesse Munea, Hyunwoo Lim, and Taeshik Shon. Network protocol fuzz testing for information systems and applications: a survey and taxonomy. *Multimedia Tools and Applications*, 75(22):14745–14757, 2016.

[249] Alessandro Murgia, Parastou Tourani, Bram Adams, and Marco Ortu. Do developers feel emotions? an exploratory analysis of emotions in software artifacts. In *Proceedings of the 11th working conference on mining software repositories*, pages 262–271, 2014.

[250] Arun Muthukkumaran, Arjun Ravichandran, Sai Shanbhag, Ramprasad Arjun, and Raghunathan Rengaswamy. Lithium-air battery electrocatalyst identification using machine learning and scibert word embeddings. In *Computer Aided Chemical Engineering*, volume 51, pages 1429–1434. Elsevier, 2022.

[251] Bhaveet Nagaria. *An investigation of human error in software development.* PhD thesis, Brunel University London, 2021.

[252] Bhaveet Nagaria and Tracy Hall. Reducing software developer human errors by improving situation awareness. *IEEE Software*, 37(6):32–37, 2020.

[253] Bhaveet Nagaria and Tracy Hall. How software developers mitigate their errors when developing code. *IEEE Transactions on Software Engineering*, 2020.

[254] Peter G. Neumann. *Computer-related risks.* Addison-Wesley Professional, 1994.

[255] Peter G. Neumann and Donald B. Parker. A summary of computer misuse techniques. In *Proceedings of the 12th National Computer Security Conference*, pages 396–407. Baltimore, MD, USA, 1989.

[256] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 315–324, 2010.

[257] Norman R. Nielsen, David H. Brandin, J.D. Madden, Brian Ruder, and G.F. Wallace. *Computer System Integrity Safeguards: System Integrity Maintenance, Final Report.* SRI International, Oct 1976. Grant No. DCR74-23774; SRI Project No. 4059.

[258] nikcheerla. nikcheerla/nooks-amd-detection-realtime. Available online at `https://huggingface.co/nikcheerla/nooks-amd-detection-realtime`, 2023.

[259] nikcheerla. nikcheerla/nooks-amd-detection-v2-full. Available online at `https://huggingface.co/nikcheerla/nooks-amd-detection-v2-full`, 2023.

[260] Nan Niu, Li Da Xu, and Zhuming Bi. Enterprise information systems architecture—analysis and evaluation. *IEEE Transactions on Industrial Informatics*, 9(4):2147–2154, 2013.

[261] Donald A. Norman. Categorization of action slips. *Psychological review*, 88(1):1, 1981.

[262] Donald A Norman. Commentary: Human error and the design of computer systems. *Communications of the ACM*, 33(1):4–7, 1990.

[263] nreimers and joaogante. distiluse-base-multilingual-cased-v1. Available online at `https://huggingface.co/sentence-transformers/distiluse-base-multilingual-cased-v1`, 2022.

[264] nreimers and joaogante. distiluse-base-multilingual-cased-v2. Available online at `https://huggingface.co/sentence-transformers/distiluse-base-multilingual-cased-v2`, 2022.

[265] nreimers and joaogante. paraphrase-multilingual-mpnet-base-v2. Available online at `https://huggingface.co/sentence-transformers/paraphrase-multilingual-mpnet-base-v2`, 2022.

[266] nreimers, joaogante, and osanseviero. paraphrase-multilingual-MiniLM-L12-v2. Available online at `https://huggingface.co/sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2`, 2022.

[267] National Institute of Standards and Technology. security. Available online at `https://csrc.nist.gov/glossary/term/security`, 2019. Accessed 2023-08-03.

[268] Elite Olshtain and Andrew Cohen. The learning of complex speech act behaviour. *TESL Canada journal*, pages 45–65, 1990.

[269] OED Online. apology, n. In *OED Online*. Oxford University Press, Oct 2021. Accessed 2021-10-28.

[270] OED Online. commission, n. In *OED Online*. Oxford University Press, Sep 2021. Accessed 2021-10-04.

[271] OED Online. omission, n. In *OED Online*. Oxford University Press, Sep 2021. Accessed 2021-10-04.

[272] Open Source Initiative. The open source definition, version 1.9. Available online at `https://opensource.org/osd`, Mar 2007. Accessed 2021-10-13.

[273] Thomas J. Ostrand and Elaine J. Weyuker. Collecting and categorizing software error data in an industrial environment. *Journal of Systems and Software*, 4(4):289–300, 1984.

[274] OWASP Foundation. OWASP top ten. `https://owasp.org/www-project-top-ten/`, Sep 2019. Accessed 2021-09-14.

[275] David N. Palacio, Daniel McCrystal, Kevin Moran, Carlos Bernal-Cárdenas, Denys Poshyvanyk, and Chris Shenefiel. Learning to identify security-related issues using convolutional neural networks. In *2019 IEEE International conference on software maintenance and evolution (ICSME)*, pages 140–144. IEEE, 2019.

[276] Yiannis Papadopoulos, John McDermid, Ralph Sasse, and Gunter Heiner. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliability Engineering & System Safety*, 71(3):229–247, 2001.

[277] Yiannis Papadopoulos, Martin Walker, M.-O. Reiser, Matthias Weber, D. Chen, Martin Törngren, David Servat, Andreas Abele, Friedhelm Stappert, H. Lonn, L. Berntsson, Rolf Johnasson, F. Tagliabo, S. Torchiaro, and Anders Sandberg. Automatic allocation of safety integrity levels. In *Proceedings of the 1st workshop on critical automotive applications: robustness & safety*, pages 7–10, 2010.

[278] Dorottya Papp, Zhendong Ma, and Levente Buttyan. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, pages 145–152. ieee, 2015.

[279] Jihun Park, Miryung Kim, Baishakhi Ray, and Doo-Hwan Bae. An empirical study of supplementary bug fixes. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 40–49. IEEE, 2012.

[280] Donald B. Parker. Computer crime: Criminal justice resource manual. *Issues and Practices in Criminal Justice*, 1989.

[281] Donald B. Parker. Computer security reference book, chapter 34, computer crime, 1992.

[282] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[283] Tekla S. Perry and Paul Wallich. Can computer crime be stopped? the proliferation of microcomputers in today's information society has brought with it new problems in protecting both computer systems and their resident intelligence. *IEEE Spectrum*, 21(5):34–45, 1984.

[284] PeterPablo. check for existence of flip(). Available online at `https://github.com/matlab2tikz/matlab2tikz/commit/4f8f3810808ce5a107d29a376768cad90a1361aa#commitcomment-8711291`, 11 2014. Accessed 2022-04-29.

[285] George A Peters and Barbara J Peters. *Human error: Causes and control.* CRC press, 2006.

[286] Venkat Pothamsetty and Bora A. Akyol. A vulnerability taxonomy for network protocols: Corresponding engineering best practice countermeasures. In *Communications, Internet, and Information Technology*, pages 168–175, 2004.

[287] Christopher Potts, Andres Suarez, and Sairam Pillai. cgpotts/swda. Available online at `https://github.com/cgpotts/swda`, 2020. Accessed 2022-04-28.

[288] Roger S. Pressman. *Software engineering: a practitioner's approach.* Palgrave macmillan, 3 edition, 1992. ISBN 0071127798.

[289] Jessica Taylor Price. Rare octopus nursery found, teeming with surprises. *National Geographic*, July 2023. Accessed 2023-08-03.

[290] Khushbakht Ali Qamar, Emre Sülün, and Eray Tüzün. Towards a taxonomy of bug tracking process smells: A quantitative analysis. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 138–147. IEEE, 2021.

[291] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[292] M.J. Ranum. Internet attacks. Presentation available online at `https://web.archive.org/web/20230720125916/http://www.ranum.com/security/computer_security/archives/internet-attacks.pdf`, 1997.

[293] Jens Rasmussen. Notes on human error analysis and prediction. In *Synthesis and analysis methods for safety and reliability studies*, pages 357–389. Springer, 1980.

[294] Jens Rasmussen. Human errors. a taxonomy for describing human malfunction in industrial installations. *Journal of occupational accidents*, 4(2-4):311–333, 1982.

[295] Jens Rasmussen. Skills, rules, and knowledge; signals, signs, and symbols, and other distinctions in human performance models. *IEEE transactions on systems, man, and cybernetics*, SMC-13(3):257–266, 1983.

[296] Jens Rasmussen. The definition of human error and a taxonomy for technical system design. In *New technology and human error*, pages 23–30. Wiley, 1987.

[297] James Reason. A framework for classifying errors. *New technology and human error*, pages 5–14, 1987.

[298] James Reason. *Human error.* Cambridge university press, Oct 1990. ISBN 0521314194.

[299] James Reason. Human error: models and management. *British Medical Journal*, 320(7237):768–770, 2000.

[300] James Reason, Antony Manstead, Stephen Stradling, James Baxter, and Karen Campbell. Errors and violations on the roads: a real distinction? *Ergonomics*, 33(10-11):1315–1332, 1990.

[301] James T. Reason and Klara Mycielska. *Absent-minded?: The psychology of mental lapses and everyday errors.* Prentice Hall, May 1982. ISBN 0130017434.

[302] Nils Reimers. Pretrained models. Available online at `https://www.sbert.net/docs/pretrained_models.html`, 2022. Accessed 2023-07-21.

[303] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[304] Jianhua Ren, I Jenkinson, Jiangping Wang, DL Xu, and JB Yang. A methodology to model causal relationships on offshore safety assessment focusing on human and organizational factors. *Journal of safety research*, 39(1): 87–100, 2008.

[305] Gabriella da Silva Rangel Ribeiro, Rafael Celestino da Silva, Márcia de Assunção Ferreira, and Grazielle Rezende da Silva. Slips, lapses and mistakes in the use of equipment by nurses in an intensive care unit. *Revista da Escola de Enfermagem da USP*, 50:0419–0426, 2016.

[306] Thomas Winfred Richardson. *The development of a database taxonomy of vulnerabilities to support the study of denial of service attacks.* Iowa State University, 2001.

[307] Tom Richardson, Jim Davis, Doug Jacobson, John Dickerson, and Laura Elkin. Developing a database of vulnerabilities to support the study of denial of service attacks. In *IEEE Symposium on Security and Privacy*, 1999.

[308] Syed Rizvi, Andrew Kurtz, Joseph Pfeffer, and Mohammad Rizvi. Securing the internet of things (iot): A security taxonomy for iot. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 163–168. IEEE, 2018.

[309] Antonio Rizzo, Sebastiano Bagnara, and Michele Visciola. Human error detection processes. *International journal of man-machine studies*, 27(5-6):555–570, 1987.

[310] Brian Roark, Margaret Mitchell, John-Paul Hosom, Kristy Hollingshead, and Jeffrey Kaye. Spoken language derived measures for detecting mild cognitive impairment. *IEEE transactions on audio, speech, and language processing*, 19(7):2081–2090, 2011.

[311] Jeffrey D. Robinson. The sequential organization of" explicit" apologies in naturally occurring english. *Research on language and social Interaction*, 37(3):291–330, 2004.

[312] Rochester Institute of Technology. Research computing services. `https://www.rit.edu/researchcomputing/`, 2023. URL `https://www.rit.edu/researchcomputing/`.

[313] Marcus K. Rogers. A new hacker taxonomy. Available online at `https://web.archive.org/web/20230720130041/https://homes.cerias.purdue.edu/~mkr/hacker.doc`, 1999. Accessed 2021-10-19.

[314] Marcus K. Rogers. *A social learning theory and moral disengagement analysis of criminal computer behavior: An exploratory study.* PhD thesis, University of Manitoba, 2001.

[315] Joanna C.S. Santos. *Toward establishing a catalog of security architecture weaknesses.* Rochester Institute of Technology, 2016.

[316] Joanna C.S. Santos, Katy Tarrit, and Mehdi Mirakhorli. A catalog of security architecture weaknesses. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 220–223. IEEE, 2017.

[317] Joanna C.S. Santos, Katy Tarrit, Adriana Sejfia, Mehdi Mirakhorli, and Matthias Galster. An empirical study of tactical vulnerabilities. *Journal of Systems and Software*, 149:263–284, 2019.

[318] Nadine B. Sarter and Heather M. Alexander. Error types and related error detection mechanisms in the aviation domain: An analysis of aviation safety reporting system incident reports. *The international journal of aviation psychology*, 10(2):189–206, 2000.

[319] Richard W. Selby Jr. Evaluations of software technologies: Testing, CLEANROOM, and metrics. Technical report, Maryland University College Park, Department of Computer Science, 1985.

[320] John W. Senders and Neville P. Moray. *Human error: Cause, prediction, and reduction.* Lawrence Erlbaum Associates, Inc, Mar 1991. ISBN 0898595983.

[321] J Bryan Sexton and Robert L Helmreich. Analyzing cockpit communications: the links between language, performance, error, and workload. *Human Performance in Extreme Environments*, 5(1):63–68, 2000.

[322] Hossain Shahriar and Mohammad Zulkernine. Taxonomy and classification of automatic monitoring of program security vulnerability exploitations. *Journal of Systems and Software*, 84(2):250–269, 2011.

[323] Alireza Shameli-Sendi, Rouzbeh Aghababaei-Barzegar, and Mohamed Cheriet. Taxonomy of information security risk assessment (isra). *Computers & security*, 57:14–30, 2016.

[324] Claude E Shannon. Programming a computer for playing chess. In *first presented at the National IRE Convention, March 9, 1949, and also in Claude Elwood Shannon Collected Papers*, pages 637–656. IEEE Press, 1993.

[325] Elizabeth Shriberg, Rebecca Bates, Paul Taylor, Andreas Stolcke, Daniel Jurafsky, Klaus Ries, Noah Coccaro, Rachel Martin, Marie Meteer, and Carol Van Ess-Dykema. Can prosody aid the automatic classification of dialog acts in conversational speech? *Language and Speech*, 41(3–4):439–487, 1998.

[326] Chris Simmons, Charles Ellis, Sajjan Shiva, Dipankar Dasgupta, and Qishi Wu. Avoidit: A cyber attack taxonomy. In *9th annual symposium on information assurance*, pages 2–12, 2014.

[327] Herbert A. Simon. A behavioral model of rational choice. *The quarterly journal of economics*, 69(1):99–118, 1955.

[328] Herbert A. Simon. Rational choice and the structure of the environment. *Psychological review*, 63(2):129, 1956.

[329] snhenson. Add heartbeat extension bounds check. `https://github.com/openssl/openssl/commit/96db9023b881d7cd9f379b0c154650d6c108e9a3`, April 2014. Accessed 2023-07-28.

[330] stackshare. GitHub. `https://stackshare.io/github`, 2021. Accessed 2021-09-23.

[331] Tor Stålhane, Torgeir Dingsøyr, Geir Kjetil Hanssen, and Nils Brede Moe. Post mortem–an assessment of two approaches. *Empirical Methods and Studies in Software Engineering: Experiences from ESERNET*, pages 129–141, 2003.

[332] Neville A. Stanton and Paul M. Salmon. Human error taxonomies applied to driving: A generic driver error taxonomy and its implications for intelligent transport systems. *Safety Science*, 47(2):227–237, 2009.

[333] Andreas Stolcke, Klaus Ries, Noah Coccaro, Elizabeth Shriberg, Rebecca Bates, Daniel Jurafsky, Paul Taylor, Rachel Martin, Marie Meteer, and Carol Van Ess-Dykema. Dialogue act modeling for automatic tagging and recognition of conversational speech. *Computational Linguistics*, 26(3):339–371, 2000.

[334] Blake E. Strom, Andy Applebaum, Doug P. Miller, Kathryn C. Nickels, Adam G. Pennington, and Cody B. Thomas. MITRE ATT&CK™: Design and philosophy. *Technical report*, 2018.

[335] Mark A Sujan, David Embrey, and Huayi Huang. On the application of human reliability analysis in healthcare: opportunities and challenges. *Reliability Engineering & System Safety*, 194:106189, 2020.

[336] Alan D. Swain and Henry E. Guttmann. Handbook of human-reliability analysis with emphasis on nuclear power plant applications: Final report. Technical report, Sandia National Laboratories, 1983.

[337] Paul L. Taylor. Beyond false positives: a typology of police shooting errors. *Criminology & Public Policy*, 18 (4):807–822, 2019.

[338] Christopher Thompson and David Wagner. A large-scale study of modern code review and security in open source projects. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 83–92, 2017.

[339] Mark J. Tiedeman. Post-mortems-methodology and experiences. *IEEE journal on selected areas in communications*, 8(2):176–180, 1990.

[340] TIOBE. TIOBE index for august 2021. Available online at `https://web.archive.org/web/20210831211029/https://www.tiobe.com/tiobe-index/`, Aug 2021. Accessed 2021-08-31.

[341] Brian M. Tomaszewski, Elizabeth A. Moore, Kyle Parnell, Alexandra M. Leader, William R. Armington, Omar Aponte, Leslie Brooks, Brienna K. Herold, Benjamin S. Meyers, Tayler Ruggero, Zachary Sutherby, Madeline Wolters, Sandy Wu, Jörg Szarzynski, Klaus Greve, and Robert Parody. Developing a geographic information capacity (gic) profile for disaster risk management under united nations framework commitments. *International journal of disaster risk reduction*, 47:101638, 2020.

[342] Sara Tonelli, Giuseppe Riccardi, Rashmi Prasad, and Aravind K. Joshi. Annotation of discourse relations for conversational spoken dialogs. In *LREC*, 2010.

[343] Louis G Tornatzky and Katherine J Klein. Innovation characteristics and innovation adoption-implementation: A meta-analysis of findings. *IEEE Transactions on engineering management*, EM-29(1):28–45, 1982.

[344] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[345] David Trepess. *A Classification Model for Human Error in Collaborative Systems*. PhD thesis, Citeseer, 2003.

[346] Anna Trosborg. Apology strategies in natives/non-natives. *Journal of pragmatics*, 11(2):147–167, 1987.

[347] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security & Privacy*, 3(6):81–84, 2005.

[348] Veronika Vincze. Uncertainty detection in natural language texts. *PhD, University of Szeged*, page 141, 2014.

[349] Gursimran Walia. Using human errors to inspect SRS. Presentation slides available online at `https://web.archive.org/web/20230720124355/https://slideplayer.com/slide/12772614/`, 2015. Accessed 2021-11-11.

[350] Gursimran Walia. Using human errors to inspect SRS. Available online at `https://web.archive.org/web/20230720125734/http://vaibhavanu.com/HEAA-TP-2016/HEAA%5C%20Training%5C%20slides.ppt`, 2018. Accessed 2022-05-02.

[351] Gursimran Singh Walia. *Using error modeling to improve and control software quality: An empirical investigation*. Mississippi State University, 2009.

[352] Gursimran Singh Walia and Jeffrey C. Carver. A systematic literature review to identify and classify software requirement errors. *Information and Software Technology*, 51(7):1087–1109, 2009.

[353] Malcolm Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science*, 141(3):53–71, 2005.

[354] Huaiqing Wang and Chen Wang. Taxonomy of security considerations and software quality. *Communications of the ACM*, 46(6):75–78, 2003.

[355] R.A. Weaver, J.A. McDermid, and T.P. Kelly. Software safety arguments: Towards a systematic categorisation of evidence. In *International System Safety Conference, Denver, CO*. Citeseer, 2002.

[356] Robert Andrew Weaver. *The safety of software: Constructing and assuring arguments*. Citeseer, 2003.

[357] Daniel James Weber. *A taxonomy of computer intrusions*. PhD thesis, Massachusetts Institute of Technology, 1998.

[358] Donald Welch and Scott Lathrop. Wireless security threat taxonomy. In *IEEE Systems, Man and Cybernetics SocietyInformation Assurance Workshop, 2003.*, pages 76–83. IEEE, 2003.

[359] Neal Whitten. *Managing software development projects formula for success*. John Wiley & Sons, Inc., 1995.

[360] Dave Wichers. OWASP top-10 2013. *OWASP Foundation, February*, 2013.

[361] Ian H Witten, Eibe Frank, and Mark A Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 3 edition, 2011.

[362] Charles Cresson Wood and William W. Banks Jr. Human error: an overlooked but significant information security problem. *Computers & Security*, 12(1):51–60, 1993.

[363] David Woods, Sidney Dekker, Richard Cook, Leila Johannesen, and Nadine Sarter. *Behind human error*. CRC Press, 2017.

[364] Peter Wright, Bob Fields, and Michael Harrison. Deriving human-error tolerance requirements from tasks. In *Proceedings of IEEE International Conference on Requirements Engineering*, pages 135–142. IEEE, 1994.

[365] Jue Wu and Wei Wang. Apology accepted: A cross-cultural study of responses to apologies by native speakers of english and chinese. *International Journal of English Linguistics*, 6(2):63–78, 2016.

[366] Zheng Yanyan and Xu Renzuo. The basic research of human factor analysis based on knowledge in software engineering. In *2008 International Conference on Computer Science and Software Engineering*, volume 5, pages 1302–1305. IEEE, 2008.

[367] Zheng Yanyan and Xu Renzuo. A human factors fault tree analysis method for software engineering. In *2008 IEEE International Conference on Industrial Engineering and Engineering Management*, pages 1971–1975. IEEE, 2008.

[368] Victor H. Yngve. A model and an hypothesis for language structure. *Proceedings of the American philosophical society*, 104(5):444–466, 1960.

[369] Weider D. Yu, Dhanya Aravind, and Passarawarin Supthaweesuk. Software vulnerability analysis for web services software systems. In *11th IEEE Symposium on Computers and Communications (ISCC'06)*, pages 740–748. IEEE, 2006.

[370] Dieter Zapf, Felix C. Brodbeck, Michael Frese, Helmut Peters, and Jochen Prümper. Errors in working with office computers: A first validation of a taxonomy for observed errors in a field setting. *International Journal of Human-Computer Interaction*, 4(4):311–339, 1992.

[371] Hossein Rouhani Zeidanloo, Mohammad Jorjor Zadeh Shooshtari, Payam Vahdani Amoli, M. Safari, and Mazdak Zamani. A taxonomy of botnet detection techniques. In *2010 3rd International Conference on Computer Science and Information Technology*, volume 2, pages 158–162. IEEE, 2010.

[372] Bin Zhang, Pei-Luen Patrick Rau, and Gavriel Salvendy. Design and evaluation of smart home user interface: effects of age, tasks and intelligence level. *Behaviour & Information Technology*, 28(3):239–249, 2009.

[373] Liang Zhang, David Choffnes, Dave Levin, Tudor Dumitraş, Alan Mislove, Aaron Schulman, and Christo Wilson. Analysis of ssl certificate reissues and revocations in the wake of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 489–502, 2014.

[374] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. Towards locating execution omission errors. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 415–424, 2007.

[375] Zhongwen Zhao and Yingchun Dai. A new method of vulnerability taxonomy based on information security attributes. In *2012 IEEE 12th International Conference on Computer and Information Technology*, pages 739–741. IEEE, 2012.

Appendices

Appendix A

Abbreviations & Acronyms

This appendix provides a listing of abbreviations and acronyms used throughout this dissertation:

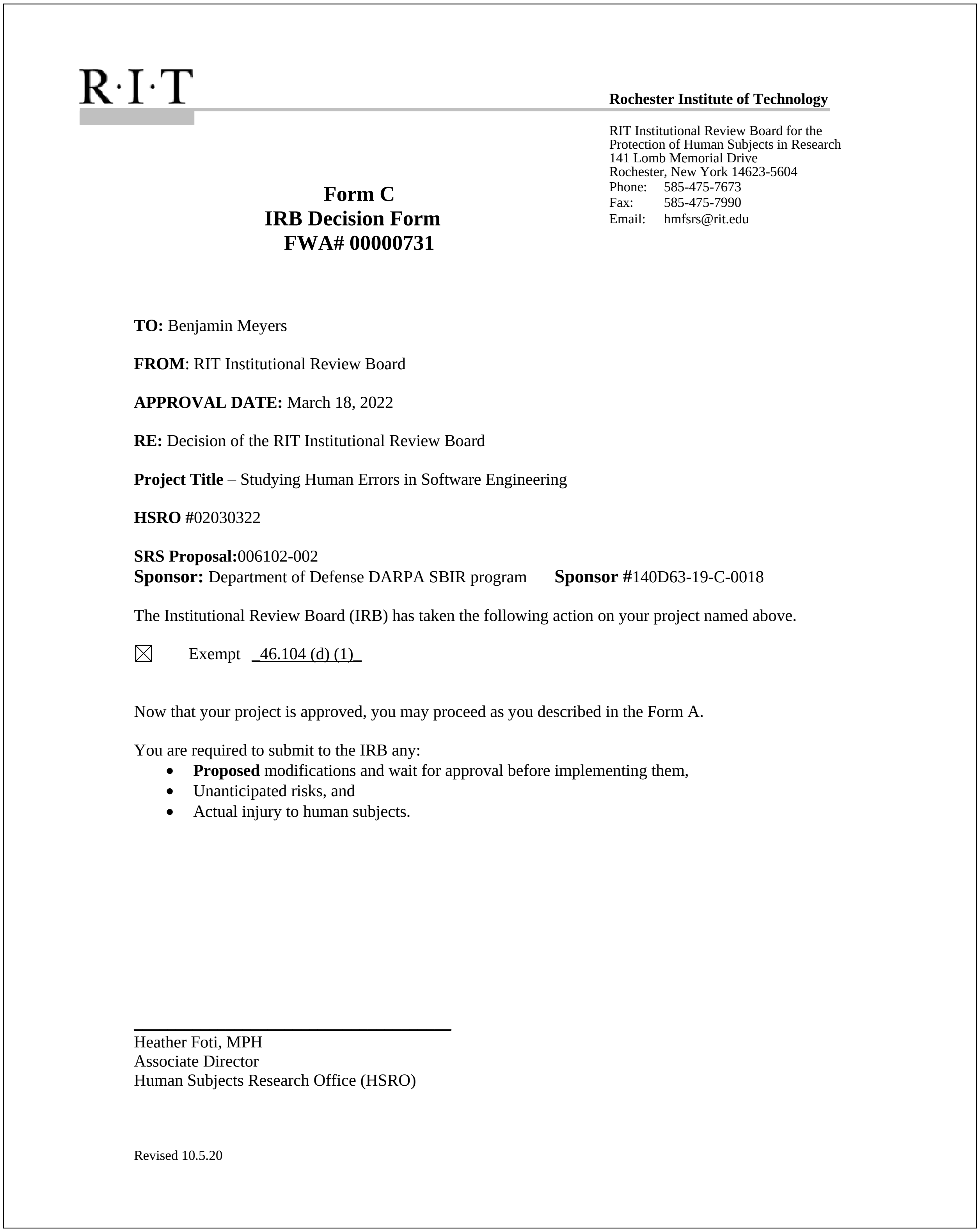
- **AER**: After-Event Review
- **API**: Application Program Interface
- **AVOIDIT**: Attack Vector, Operational Impact, Defense, Information Impact, and Target (from Simmons *et al.* [326])
- **BERT**: Bidirectional Encoder Representations from Transformers (from Devlin *et al.* [77] and Reimers & Gurevych [303])
- **CAPEC**: Common Attack Pattern Enumeration and Classification (from [233])
- **CERT**: Community Emergency Response Team (from Howard [136])
- **CLI**: Command Line Interface
- **CPU**: Central Processing Unit
- **CVE**: Common Vulnerabilities and Exposures [237]
- **CVSS**: Common Vulnerability Scoring System (from [207])
- **CWE**: Common Weakness Enumeration [235]
- **CWSS**: Common Weakness Scoring System (from [201])
- **Dev-HET**: Developer Human Error Taxonomy (from Anu *et al.* [20])
- **DNS**: Domain Name System
- **DoS**: Denial-of-Service
- **FN**: False Negative
- **FP:** False Positive
- **GEMS**: Generic Error-Modelling System (from Reason [300])
- **HE**: Human Error
- **HEA**: Human Error Assessment
- **HEAA**: Human Error Abstraction Assist (from Anu *et al.* [14])
- **H.E.R.E.**: Human Error Reflection Engine
- **HET**: Human Error Taxonomy (from Anu *et al.* [12, 15, 18, 22, 23])
- **HRA**: Human Reliability Analysis
- **HTTP**: Hypertext Transfer Protocol
- **ICMP**: Internet Control Message Protocol

- **IDE**: Integrated Development Environment
- **IDS**: Intrusion Detection System
- **IoT**: Internet-of-Things
- **JS**: JavaScript
- **KB**: Knowledge-Based
- **MLM**: Masked Language Modeling
- **NLP**: Natural Language Processing
- **NSP**: Next Sentence Prediction
- **NVD**: National Vulnerability Database
- **O/C**: Omission/Commission
- **ODC**: Orthogonal Defect Classification
- **OODA**: Observe, Orient, Decide, Act (from Nagaria & Hall [251, 252])
- **OS**: Operating System
- **OWASP**: Open Web Application Security Project (from [274])
- **Pr.**: Precision
- **PA**: Protection Analysis (from Bisbey & Hollingworth [41])
- **POS**: Parts-of-Speech
- **Re.**: Recall
- **RB**: Rule-Based
- **RCA**: Root Cause Analysis
- **RET**: Requirement Error Taxonomy (from Walia *et al.* [351, 352])
- **RISOS**: Research in Secured Operating Systems (from Abbott *et al.* [1])
- **RIT**: Rochester Institute of Technology
- **SB**: Skill-Based
- **SLR**: Systematic Literature Review
- **SRK**: Skill-Rule-Knowledge (from Rasmussen [293, 295])
- **SSH**: Secure Shell
- **T.H.E.S.E.**: Taxonomy of Human Errors in Software Engineering
- **TN**: True Negative
- **TP**: True Positive
- **TSSD**: Taxonomy System of Software Developers (from Huang *et al.* [143])
- **VERDICT**: Validation Exposure Randomness Deallocation Improper Conditions Taxonomy (from Lough [192])
- **VHP**: Vulnerability History Project
- **WE**: Word Embeddings (from Mikolov *et al.* [222])
- **XSS**: Cross-Site Scripting

Appendix B

Institutional Review Board Approval

Institutional Review Board (IRB) approval for our research involving human subjects (Section 5.2) was granted by the Human Subjects Research Office at the Rochester Institute of Technology on March 18, 2022. Figure B.1 shows the IRB decision form granting approval to this research.



R·I·T

Rochester Institute of Technology

RIT Institutional Review Board for the
Protection of Human Subjects in Research
141 Lomb Memorial Drive
Rochester, New York 14623-5604
Phone: 585-475-7673
Fax: 585-475-7990
Email: hmfsrs@rit.edu

Form C
IRB Decision Form
FWA# 00000731

TO: Benjamin Meyers

FROM: RIT Institutional Review Board

APPROVAL DATE: March 18, 2022

RE: Decision of the RIT Institutional Review Board

Project Title – Studying Human Errors in Software Engineering

HSRO #02030322

SRS Proposal:006102-002
Sponsor: Department of Defense DARPA SBIR program **Sponsor** #140D63-19-C-0018

The Institutional Review Board (IRB) has taken the following action on your project named above.

☒ Exempt _46.104 (d) (1)_

Now that your project is approved, you may proceed as you described in the Form A.

You are required to submit to the IRB any:

- **Proposed** modifications and wait for approval before implementing them,
- Unanticipated risks, and
- Actual injury to human subjects.

Heather Foti, MPH
Associate Director
Human Subjects Research Office (HSRO)

Revised 10.5.20

Figure B.1: Institutional Review Board Approval

Form C (IRB Decision Form) from the Institutional Review Board at Rochester Institute of Technology. Signature redacted.

Appendix C

Informed Consent

For the user study described in Section 5.2, we obtained informed consent from participants prior to their participation. Figure C.1 and Figure C.2 show the informed consent text provided to participants for Session 1 and Session 2, respectively.

The purpose of this research is to elicit experiences with human error while refining a Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.). Human error is an emerging field of study in software engineering and initial results indicate that learning about and reflecting upon human errors positively impacts software engineering activities.

If you choose to participate in this research, you will be given an introduction to human error and T.H.E.S.E. and then tasked with filling out a Google Form for each human error you experience throughout the study. Each week, you will meet for 30 minutes with Benjamin Meyers to discuss the human errors you have experienced during the week and answer some questions. You will also be tasked with manually categorizing some human errors from GitHub according to T.H.E.S.E.

Participation in this research is **entirely voluntary**. To participate, you must be (1) a current student or recent graduate of a computing program at RIT and (2) actively developing software every week for the duration of the research project. The project will span 4 weeks, and at the end you will receive a **$75 gift card**. You must complete all of the required tasks in order to receive the gift card.

In summary, the time commitment for this research is about 7-8 hours (60-120 minutes per week), with the following activities:

(1) 30 minutes, once: brief introduction to human error and T.H.E.S.E.

(2) 30 minutes, weekly: documenting the human errors you experience

(3) 30 minutes, weekly: discussion with Ben Meyers

(4) 1-2 hours, once: categorizing human errors from GitHub

(5) 30 minutes, once: completing a brief survey at the end of the research project

If you choose to participate in this research, you acknowledge that **your survey responses may be published** as part of academic research. Your name and any other information that could be used to identify you will **never** be published. Please note that your weekly meetings will be recorded, and that quotes (without attribution) may be published, but audio/video recordings will **never** be published.

If you have any questions while filling out this form, please email Benjamin Meyers: bsm9339@rit.edu.

Figure C.1: Informed Consent Text for Phase 1 Participants

Study participants digitally signed this form, verifying their understanding of their responsibilities and how their data would be used.

The purpose of this research is to elicit experiences with human error while refining a Taxonomy of Human Errors in Software Engineering (T.H.E.S.E.). Human error is an emerging field of study in software engineering and initial results indicate that learning about and reflecting upon human errors positively impacts software engineering activities.

If you choose to participate in this research, you will be given an introduction to human error and T.H.E.S.E. and then tasked with filling out a Google Form for each human error you experience throughout the study. Each week, you will meet for 30 minutes with Benjamin Meyers to discuss the human errors you have experienced during the week and answer some questions.

Participation in this research is **entirely voluntary**. To participate, you must be a SWEN-561 student. Alternatively, we are open to accepting RIT students who are actively developing software (either through coursework or on co-op). The project will span 8 weeks, and at the end you will receive a **$75 gift card**. You must complete all of the required tasks in order to receive the gift card.

In summary, the time commitment for this research is about 8 hours (1 hour per week), with the following activities:

(1) 30 minutes, once: brief introduction to human error and T.H.E.S.E.

(2) 30 minutes, weekly: documenting the human errors you experience

(3) 30 minutes, weekly: discussion with Ben Meyers

(4) 30 minutes, once: completing a brief survey at the end of the research project

If you choose to participate in this research, you acknowledge that **your survey responses may be published** as part of academic research. Your name and any other information that could be used to identify you will **never** be published. Please note that your weekly meetings will be recorded, and that quotes (without attribution) may be published, but audio/video recordings will **never** be published.

If you have any questions while filling out this form, please email Benjamin Meyers: bsm9339@rit.edu.

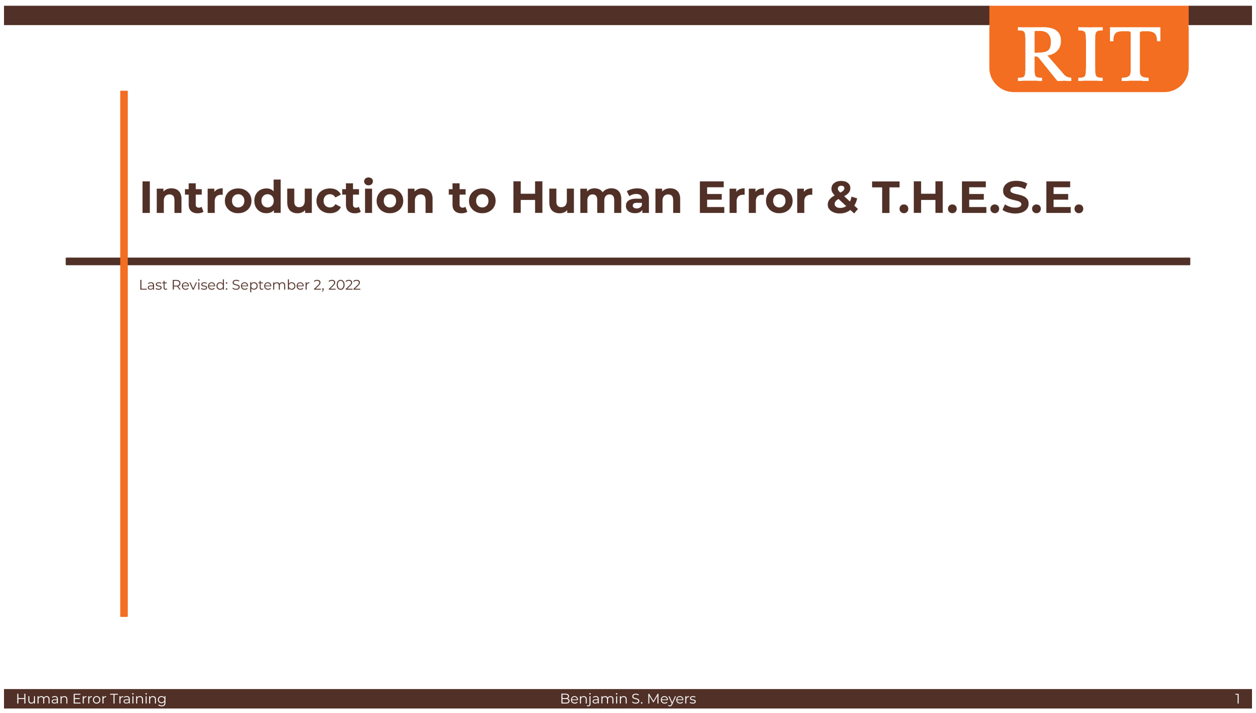
Figure C.2: Informed Consent Text for Phase 2 Participants

Study participants digitally signed this form, verifying their understanding of their responsibilities and how their data would be used. A few minor changes occurred from Figure C.1: (1) categorizing human errors from GitHub removed from responsibilities, (2) participation criteria changed, (3) time commitment changed.

Appendix D

Human Error Training Slides

This appendix provides the slides used during human error training in our user study, as described in Section 5.2.



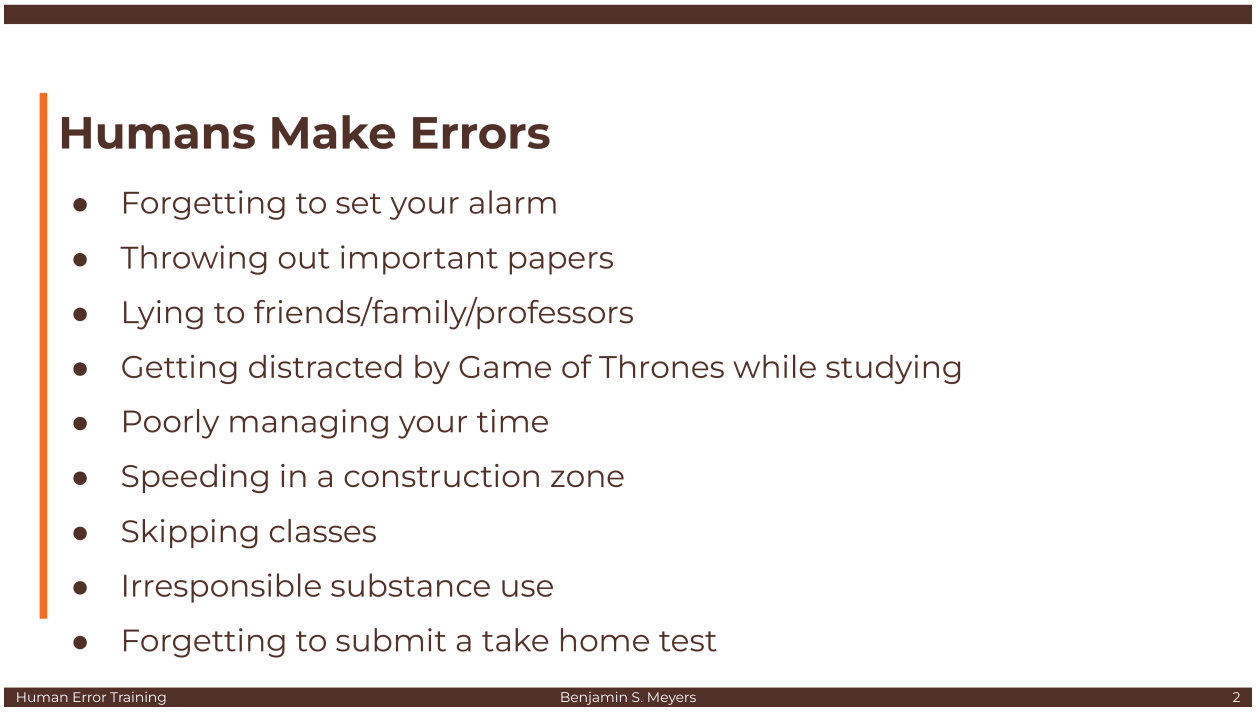
RIT

Introduction to Human Error & T.H.E.S.E.

Last Revised: September 2, 2022

Human Error Training Benjamin S. Meyers 1

(a) Slide 1: Title slide



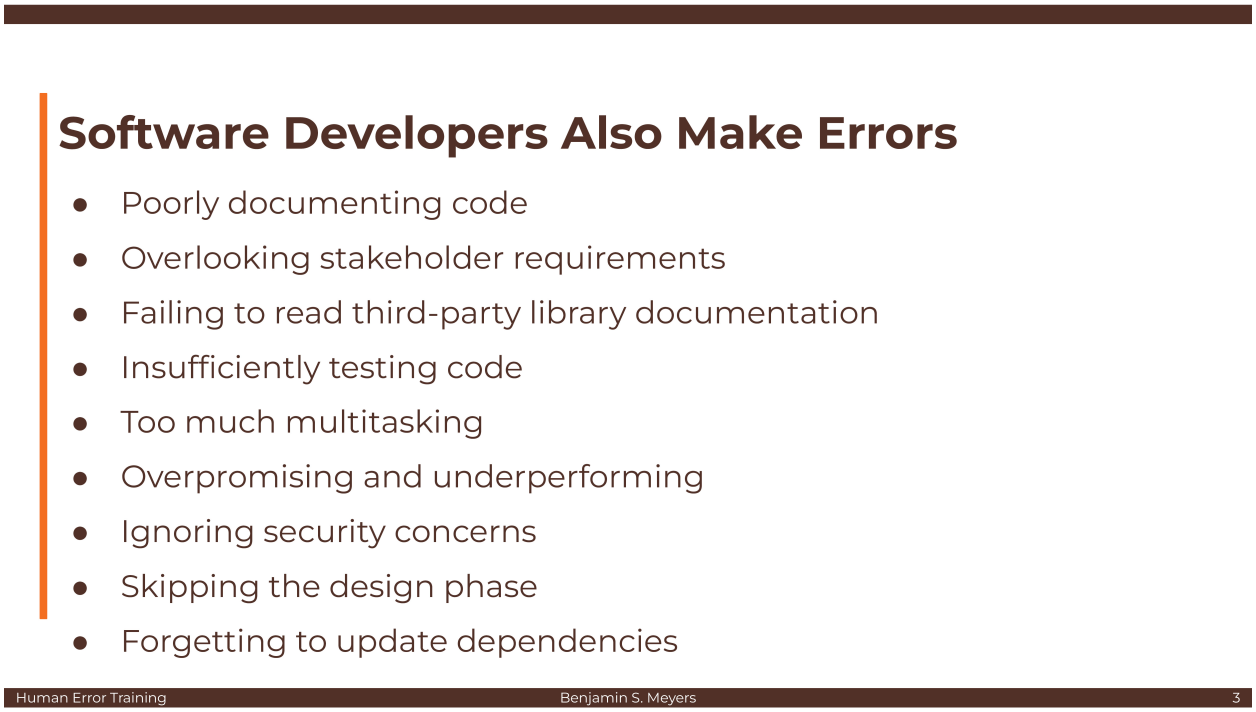
Humans Make Errors

- Forgetting to set your alarm
- Throwing out important papers
- Lying to friends/family/professors
- Getting distracted by Game of Thrones while studying
- Poorly managing your time
- Speeding in a construction zone
- Skipping classes
- Irresponsible substance use
- Forgetting to submit a take home test

Human Error Training Benjamin S. Meyers 2

(b) Slide 2: Introduction with common errors that students might make

Figure D.1: Human Error Training Slides

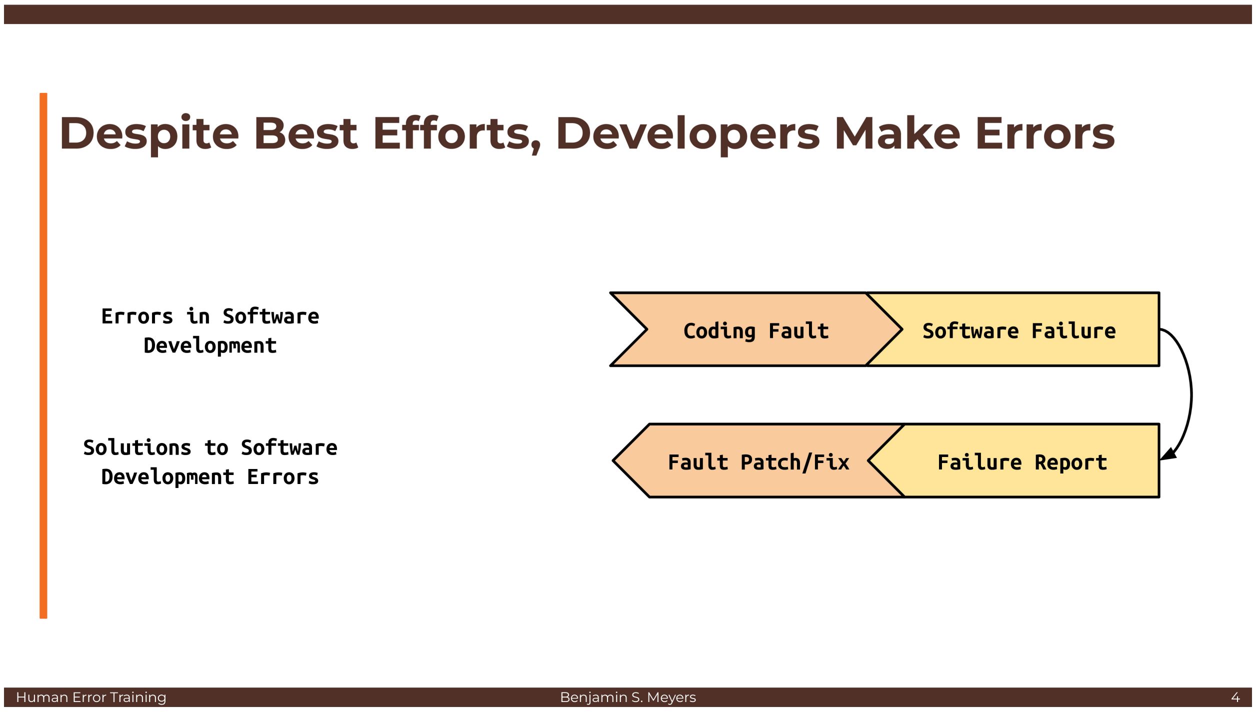


Software Developers Also Make Errors

- Poorly documenting code
- Overlooking stakeholder requirements
- Failing to read third-party library documentation
- Insufficiently testing code
- Too much multitasking
- Overpromising and underperforming
- Ignoring security concerns
- Skipping the design phase
- Forgetting to update dependencies

Human Error Training Benjamin S. Meyers 3

(c) Slide 3: Discussion of common errors in software engineering



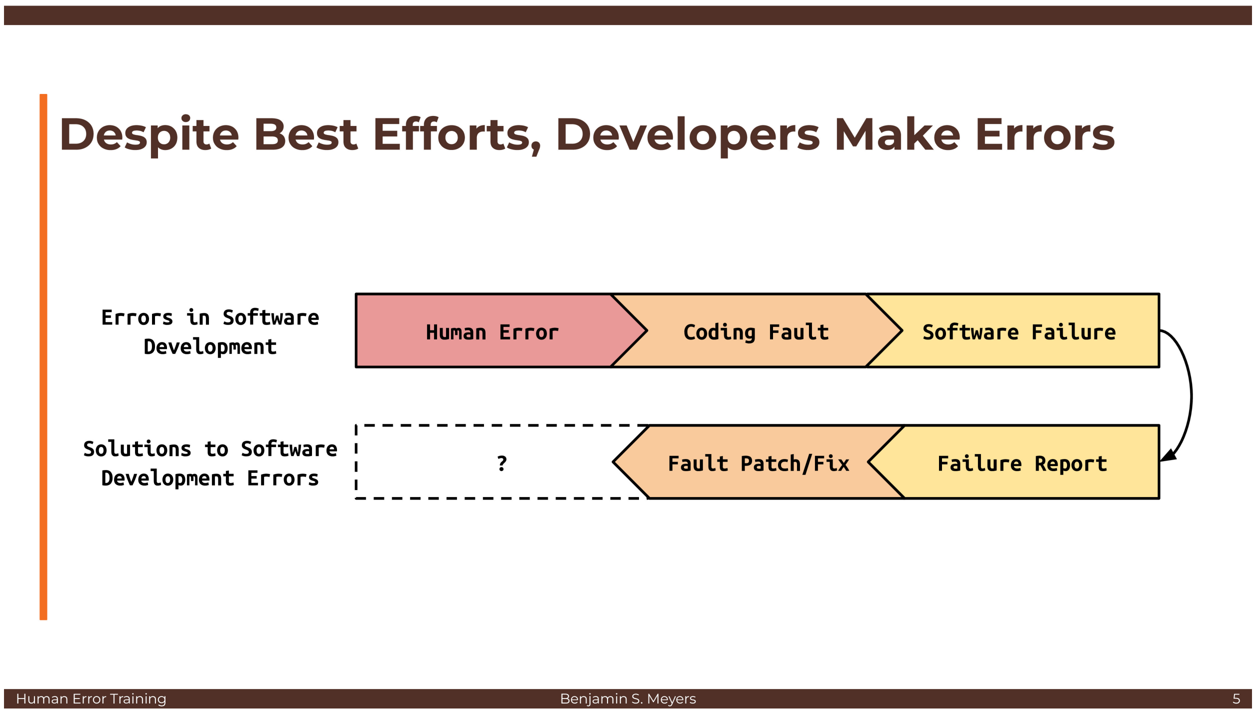
Despite Best Efforts, Developers Make Errors

Errors in Software Development
Solutions to Software Development Errors
Coding Fault
Software Failure
Failure Report
Fault Patch/Fix

Human Error Training Benjamin S. Meyers 4

(d) Slide 4: Illustration of typical error lifecycle in software engineering

Figure D.1: Human Error Training Slides (Continued)

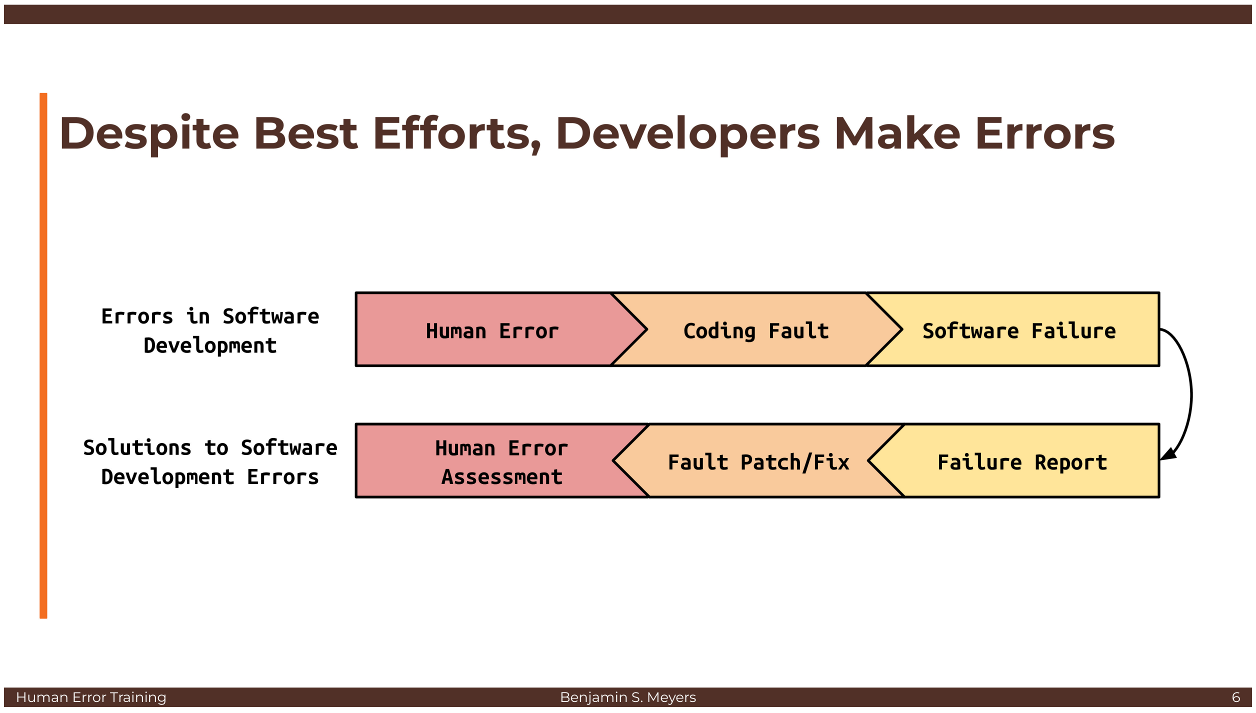


Despite Best Efforts, Developers Make Errors

Errors in Software Development
Human Error
Coding Fault
Software Failure
Solutions to Software Development Errors
?
Fault Patch/Fix
Failure Report

Human Error Training Benjamin S. Meyers 5

(e) Slide 5: Introduction to the notion of human errors leading to coding faults



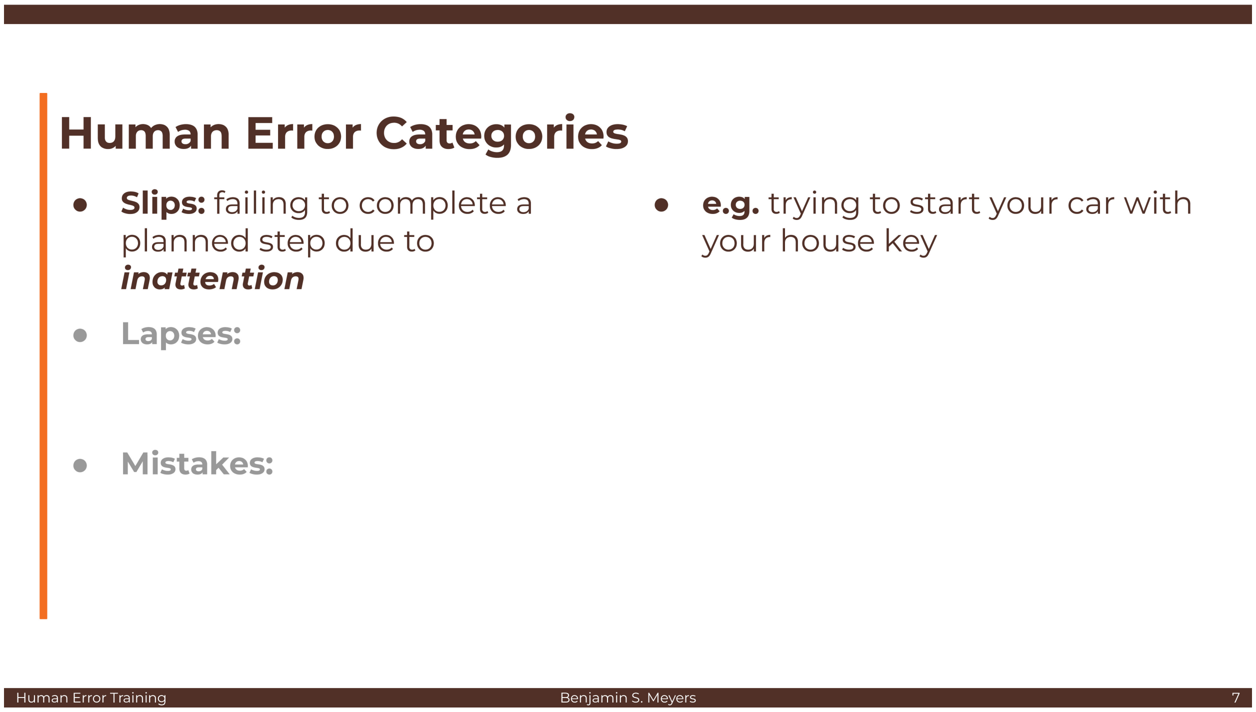
Despite Best Efforts, Developers Make Errors

Errors in Software Development
Human Error
Coding Fault
Software Failure
Solutions to Software Development Errors
Human Error Assessment
Fault Patch/Fix
Failure Report

Human Error Training Benjamin S. Meyers 6

(f) Slide 6: Discussion of the goal of human error assessment in software engineering

Figure D.1: Human Error Training Slides (Continued)



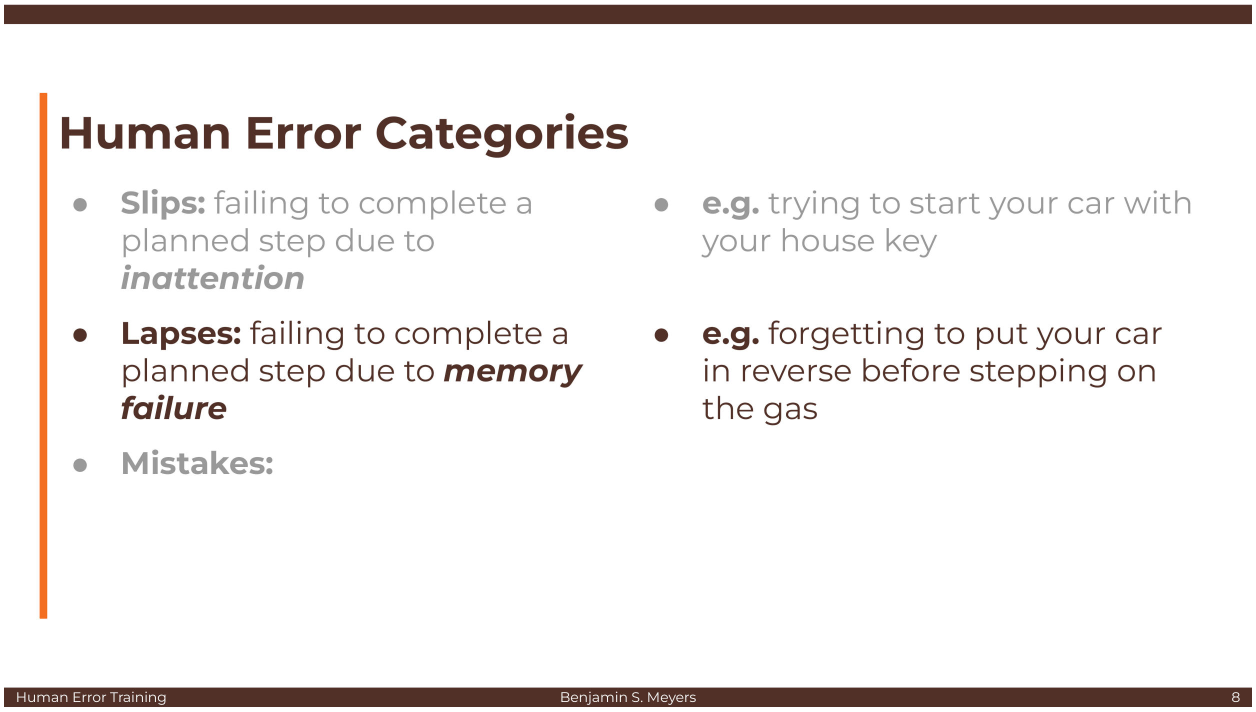
Human Error Categories

- **Slips:** failing to complete a planned step due to ***inattention***
- **Lapses:**
- **Mistakes:**

- **e.g.** trying to start your car with your house key

Human Error Training Benjamin S. Meyers 7

(g) Slide 7: Introduction to slips with definition and examples



Human Error Categories

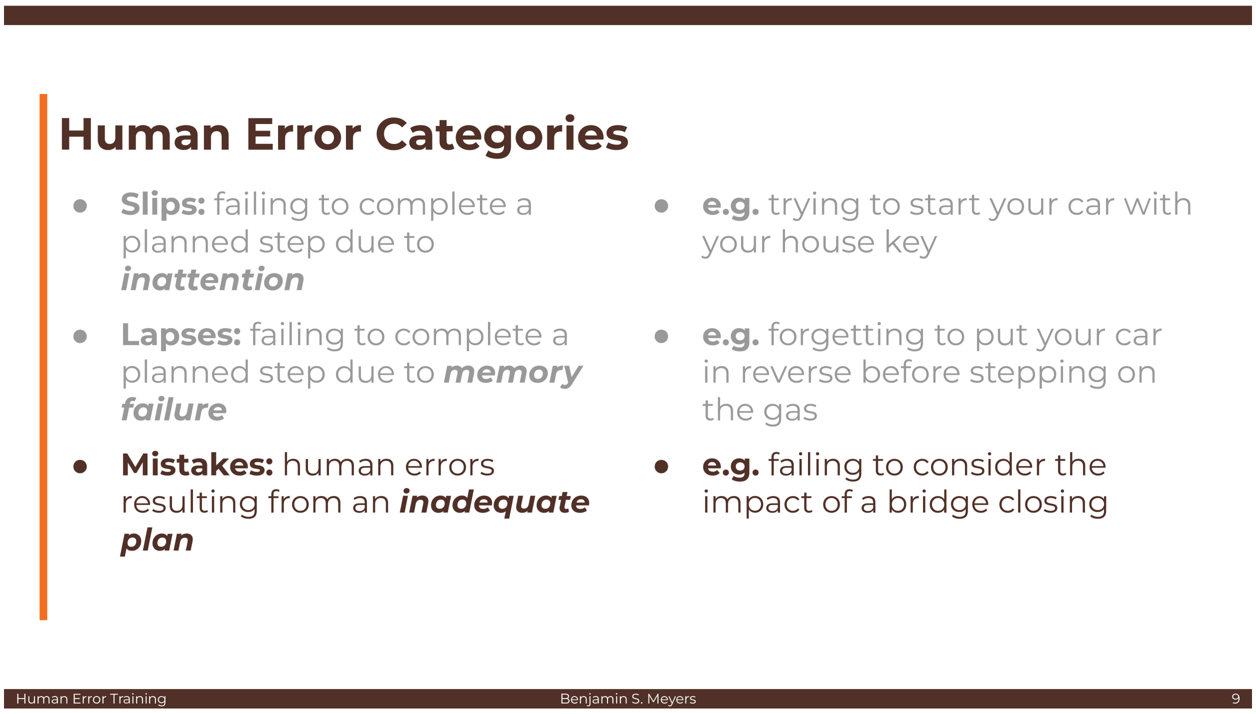
- **Slips:** failing to complete a planned step due to ***inattention***
- **Lapses:** failing to complete a planned step due to ***memory failure***
- **Mistakes:**

- **e.g.** trying to start your car with your house key
- **e.g.** forgetting to put your car in reverse before stepping on the gas

Human Error Training Benjamin S. Meyers 8

(h) Slide 8: Introduction to lapses with definition and examples

Figure D.1: Human Error Training Slides (Continued)

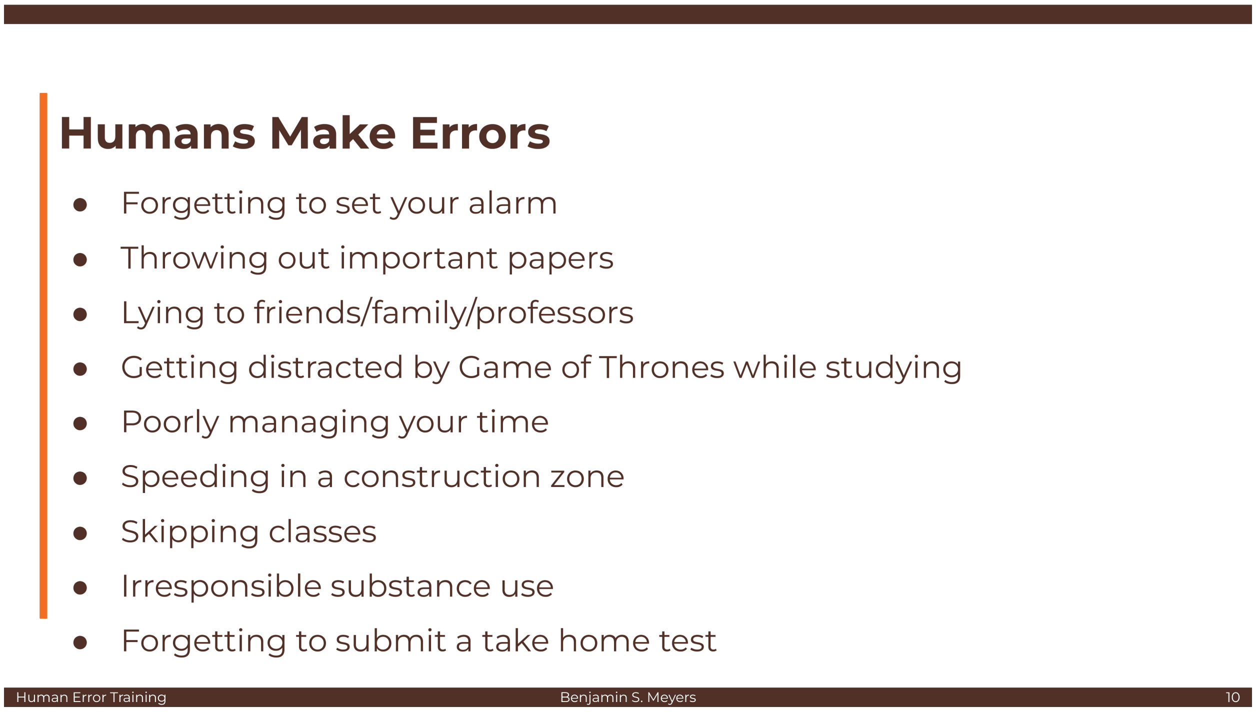


Human Error Categories

- **Slips:** failing to complete a planned step due to ***inattention***
- **e.g.** trying to start your car with your house key
- **Lapses:** failing to complete a planned step due to ***memory failure***
- **e.g.** forgetting to put your car in reverse before stepping on the gas
- **Mistakes:** human errors resulting from an ***inadequate plan***
- **e.g.** failing to consider the impact of a bridge closing

Human Error Training Benjamin S. Meyers 9

(i) Slide 9: Introduction to mistakes with definition and examples



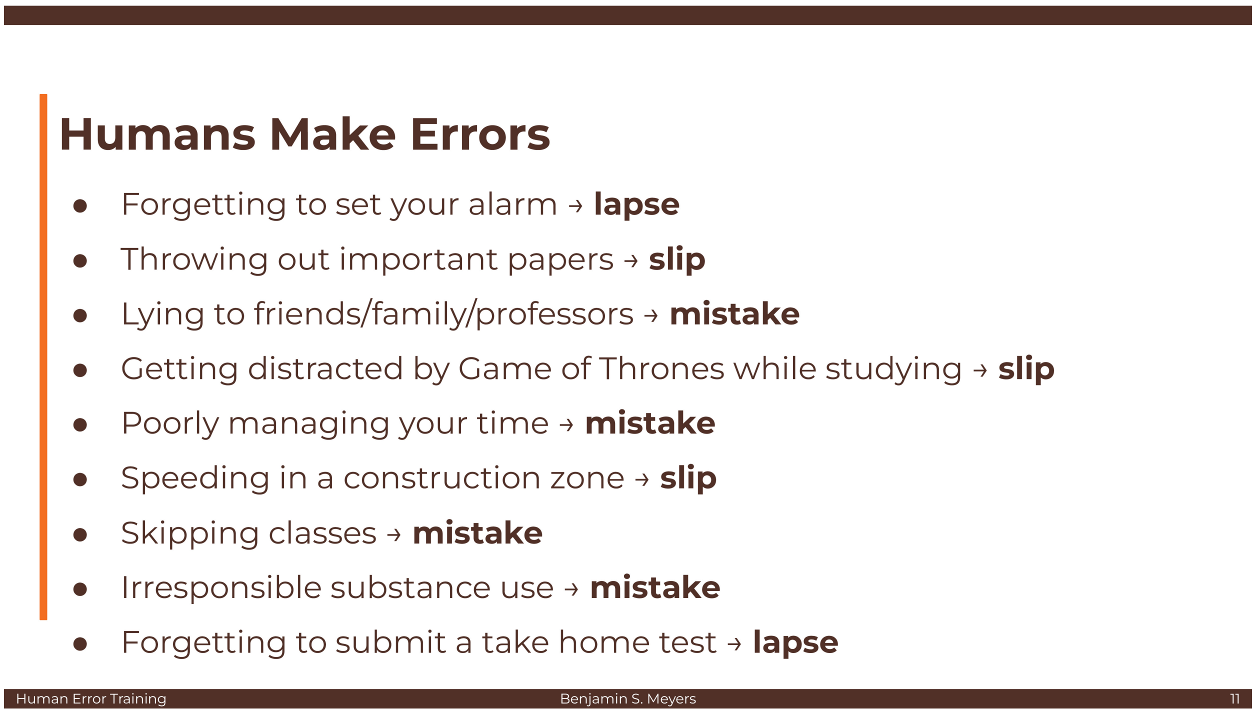
Humans Make Errors

- Forgetting to set your alarm
- Throwing out important papers
- Lying to friends/family/professors
- Getting distracted by Game of Thrones while studying
- Poorly managing your time
- Speeding in a construction zone
- Skipping classes
- Irresponsible substance use
- Forgetting to submit a take home test

Human Error Training Benjamin S. Meyers 10

(j) Slide 10: Return to examples of typical student human errors

Figure D.1: Human Error Training Slides (Continued)

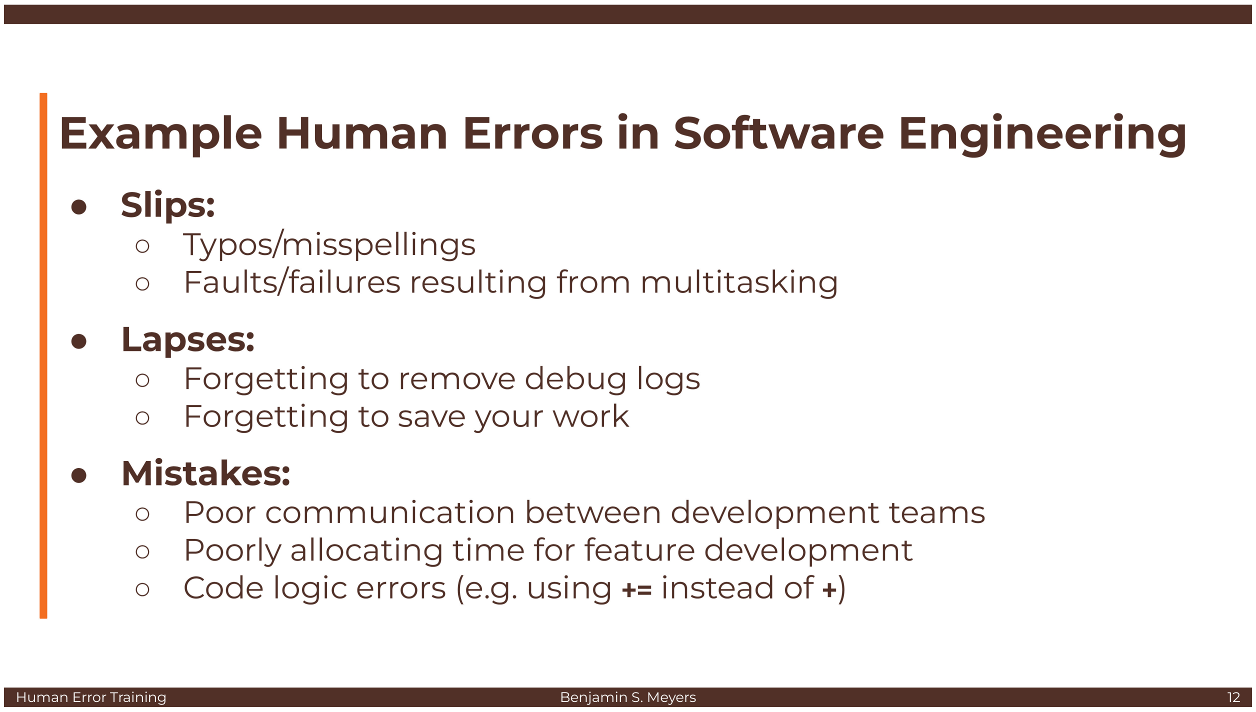


Humans Make Errors

- Forgetting to set your alarm → **lapse**
- Throwing out important papers → **slip**
- Lying to friends/family/professors → **mistake**
- Getting distracted by Game of Thrones while studying → **slip**
- Poorly managing your time → **mistake**
- Speeding in a construction zone → **slip**
- Skipping classes → **mistake**
- Irresponsible substance use → **mistake**
- Forgetting to submit a take home test → **lapse**

Human Error Training Benjamin S. Meyers 11

(k) Slide 11: Discussion of typical student human errors in context



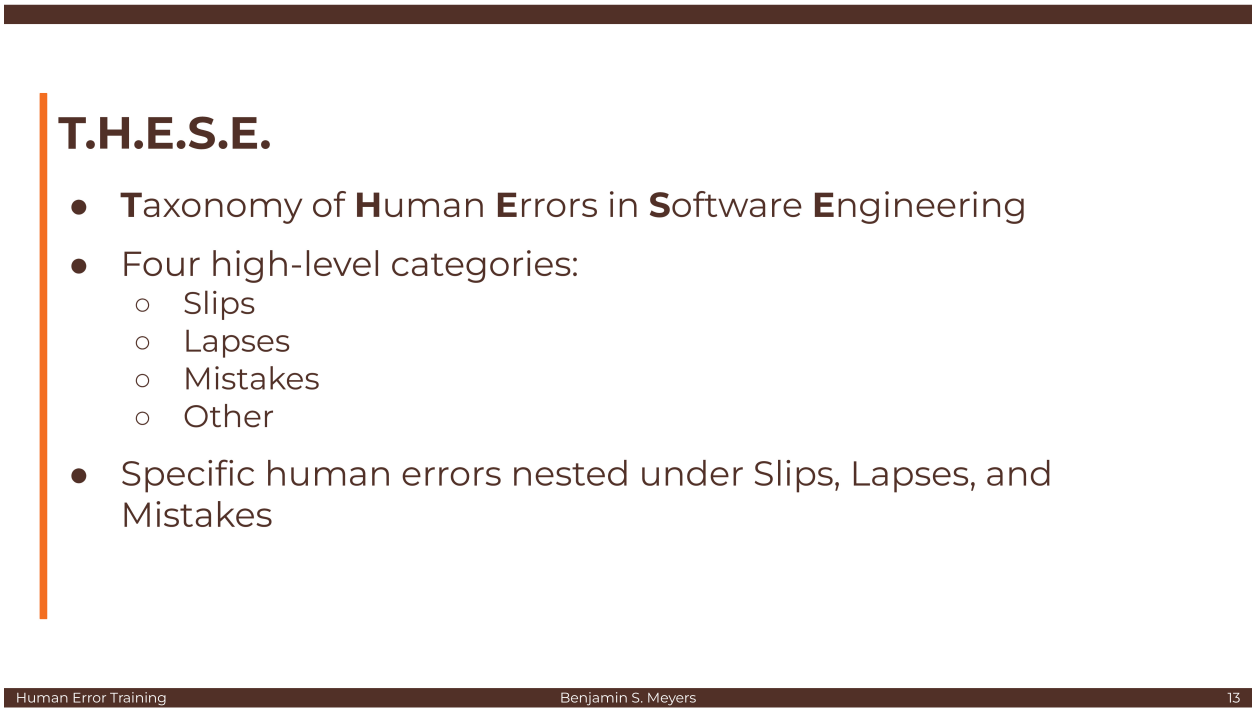
Example Human Errors in Software Engineering

- **Slips:**
  - Typos/misspellings
  - Faults/failures resulting from multitasking
- **Lapses:**
  - Forgetting to remove debug logs
  - Forgetting to save your work
- **Mistakes:**
  - Poor communication between development teams
  - Poorly allocating time for feature development
  - Code logic errors (e.g. using **+=** instead of **+**)

Human Error Training Benjamin S. Meyers 12

(l) Slide 12: More examples of human errors in software engineering

Figure D.1: Human Error Training Slides (Continued)

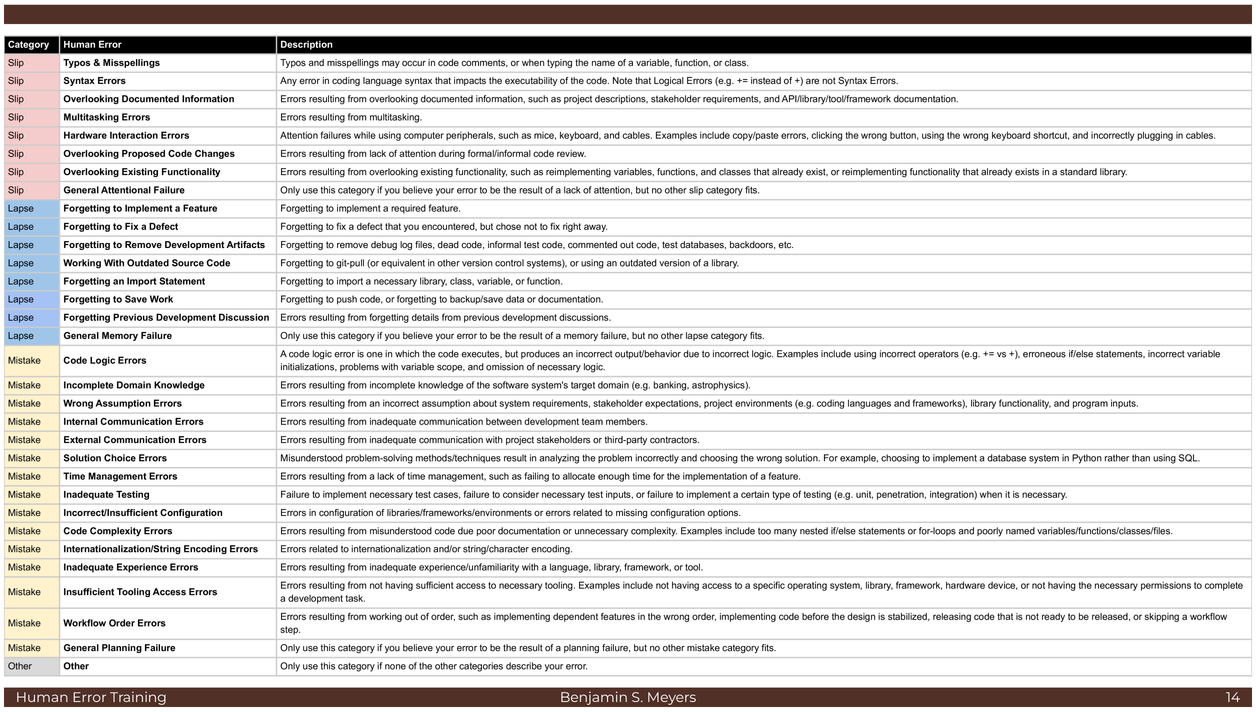


T.H.E.S.E.

- **T**axonomy of **H**uman **E**rrors in **S**oftware **E**ngineering
- Four high-level categories:
  - Slips
  - Lapses
  - Mistakes
  - Other
- Specific human errors nested under Slips, Lapses, and Mistakes

Human Error Training Benjamin S. Meyers 13

(m) Slide 13: Introduction to T.H.E.S.E.

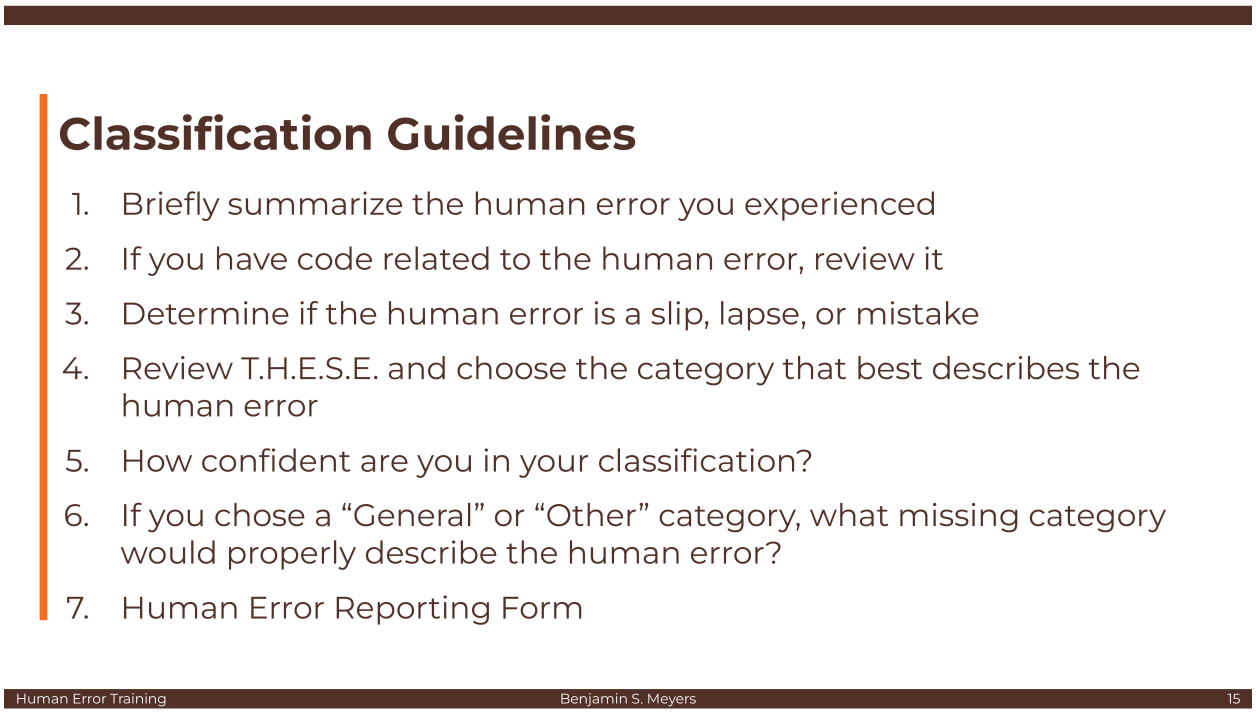


Category	Human Error	Description
Slip	**Typos & Misspellings**	Typos and misspellings may occur in code comments, or when typing the name of a variable, function, or class.
Slip	**Syntax Errors**	Any error in coding language syntax that impacts the executability of the code. Note that Logical Errors (e.g. += instead of +) are not Syntax Errors.
Slip	**Overlooking Documented Information**	Errors resulting from overlooking documented information, such as project descriptions, stakeholder requirements, and API/library/tool/framework documentation.
Slip	**Multitasking Errors**	Errors resulting from multitasking.
Slip	**Hardware Interaction Errors**	Attention failures while using computer peripherals, such as mice, keyboard, and cables. Examples include copy/paste errors, clicking the wrong button, using the wrong keyboard shortcut, and incorrectly plugging in cables.
Slip	**Overlooking Proposed Code Changes**	Errors resulting from lack of attention during formal/informal code review.
Slip	**Overlooking Existing Functionality**	Errors resulting from overlooking existing functionality, such as reimplementing variables, functions, and classes that already exist, or reimplementing functionality that already exists in a standard library.
Slip	**General Attentional Failure**	Only use this category if you believe your error to be the result of a lack of attention, but no other slip category fits.
Lapse	**Forgetting to Implement a Feature**	Forgetting to implement a required feature.
Lapse	**Forgetting to Fix a Defect**	Forgetting to fix a defect that you encountered, but chose not to fix right away.
Lapse	**Forgetting to Remove Development Artifacts**	Forgetting to remove debug log files, dead code, informal test code, commented out code, test databases, backdoors, etc.
Lapse	**Working With Outdated Source Code**	Forgetting to git-pull (or equivalent in other version control systems), or using an outdated version of a library.
Lapse	**Forgetting an Import Statement**	Forgetting to import a necessary library, class, variable, or function.
Lapse	**Forgetting to Save Work**	Forgetting to push code, or forgetting to backup/save data or documentation.
Lapse	**Forgetting Previous Development Discussion**	Errors resulting from forgetting details from previous development discussions.
Lapse	**General Memory Failure**	Only use this category if you believe your error to be the result of a memory failure, but no other lapse category fits.
Mistake	**Code Logic Errors**	A code logic error is one in which the code executes, but produces an incorrect output/behavior due to incorrect logic. Examples include using incorrect operators (e.g. += vs +), erroneous if/else statements, incorrect variable initializations, problems with variable scope, and omission of necessary logic.
Mistake	**Incomplete Domain Knowledge**	Errors resulting from incomplete knowledge of the software system's target domain (e.g. banking, astrophysics).
Mistake	**Wrong Assumption Errors**	Errors resulting from an incorrect assumption about system requirements, stakeholder expectations, project environments (e.g. coding languages and frameworks), library functionality, and program inputs.
Mistake	**Internal Communication Errors**	Errors resulting from inadequate communication between development team members.
Mistake	**External Communication Errors**	Errors resulting from inadequate communication with project stakeholders or third-party contractors.
Mistake	**Solution Choice Errors**	Misunderstood problem-solving methods/techniques result in analyzing the problem incorrectly and choosing the wrong solution. For example, choosing to implement a database system in Python rather than using SQL.
Mistake	**Time Management Errors**	Errors resulting from a lack of time management, such as failing to allocate enough time for the implementation of a feature.
Mistake	**Inadequate Testing**	Failure to implement necessary test cases, failure to consider necessary test inputs, or failure to implement a certain type of testing (e.g. unit, penetration, integration) when it is necessary.
Mistake	**Incorrect/Insufficient Configuration**	Errors in configuration of libraries/frameworks/environments or errors related to missing configuration options.
Mistake	**Code Complexity Errors**	Errors resulting from misunderstood code due poor documentation or unnecessary complexity. Examples include too many nested if/else statements or for-loops and poorly named variables/functions/classes/files.
Mistake	**Internationalization/String Encoding Errors**	Errors related to internationalization and/or string/character encoding.
Mistake	**Inadequate Experience Errors**	Errors resulting from inadequate experience/unfamiliarity with a language, library, framework, or tool.
Mistake	**Insufficient Tooling Access Errors**	Errors resulting from not having sufficient access to necessary tooling. Examples include not having access to a specific operating system, library, framework, hardware device, or not having the necessary permissions to complete a development task.
Mistake	**Workflow Order Errors**	Errors resulting from working out of order, such as implementing dependent features in the wrong order, implementing code before the design is stabilized, releasing code that is not ready to be released, or skipping a workflow step.
Mistake	**General Planning Failure**	Only use this category if you believe your error to be the result of a planning failure, but no other mistake category fits.
Other	**Other**	Only use this category if none of the other categories describe your error.

Human Error Training Benjamin S. Meyers 14

(n) Slide 14: Full listing of T.H.E.S.E. categories

Figure D.1: Human Error Training Slides (Continued)

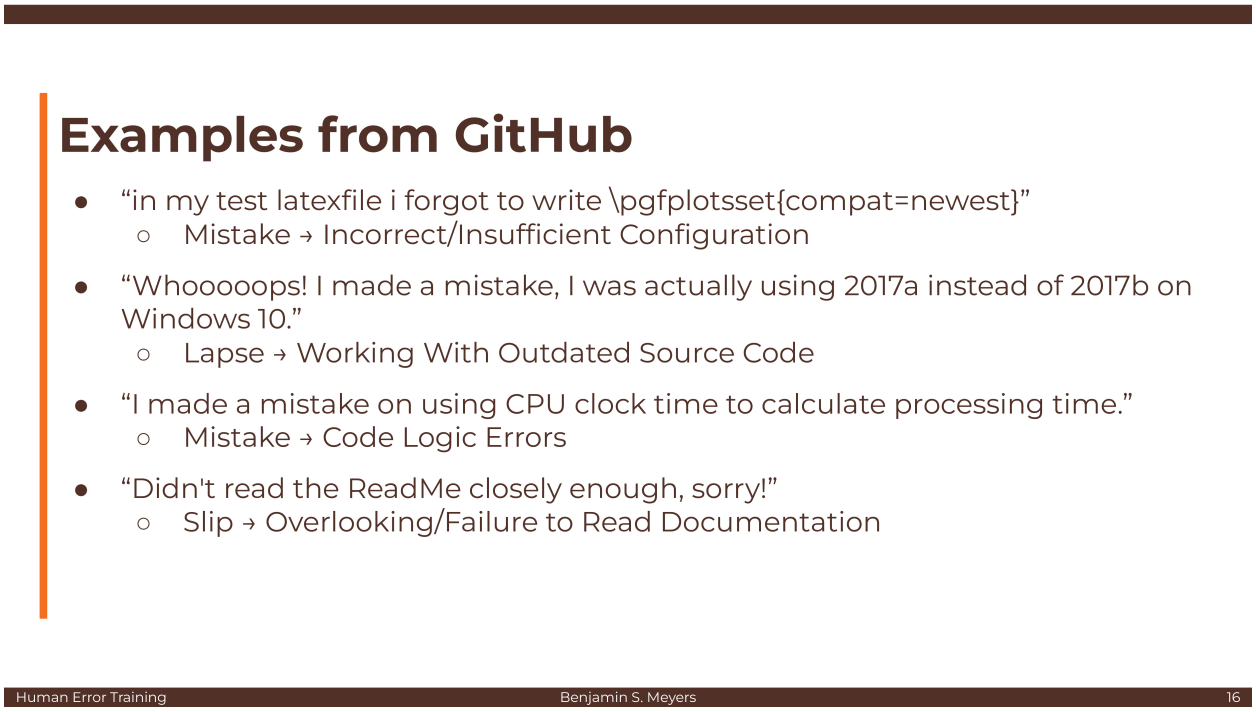


Classification Guidelines

1. Briefly summarize the human error you experienced
2. If you have code related to the human error, review it
3. Determine if the human error is a slip, lapse, or mistake
4. Review T.H.E.S.E. and choose the category that best describes the human error
5. How confident are you in your classification?
6. If you chose a "General" or "Other" category, what missing category would properly describe the human error?
7. Human Error Reporting Form

Human Error Training Benjamin S. Meyers 15

(o) Slide 15: Guidelines for human error classification using T.H.E.S.E.



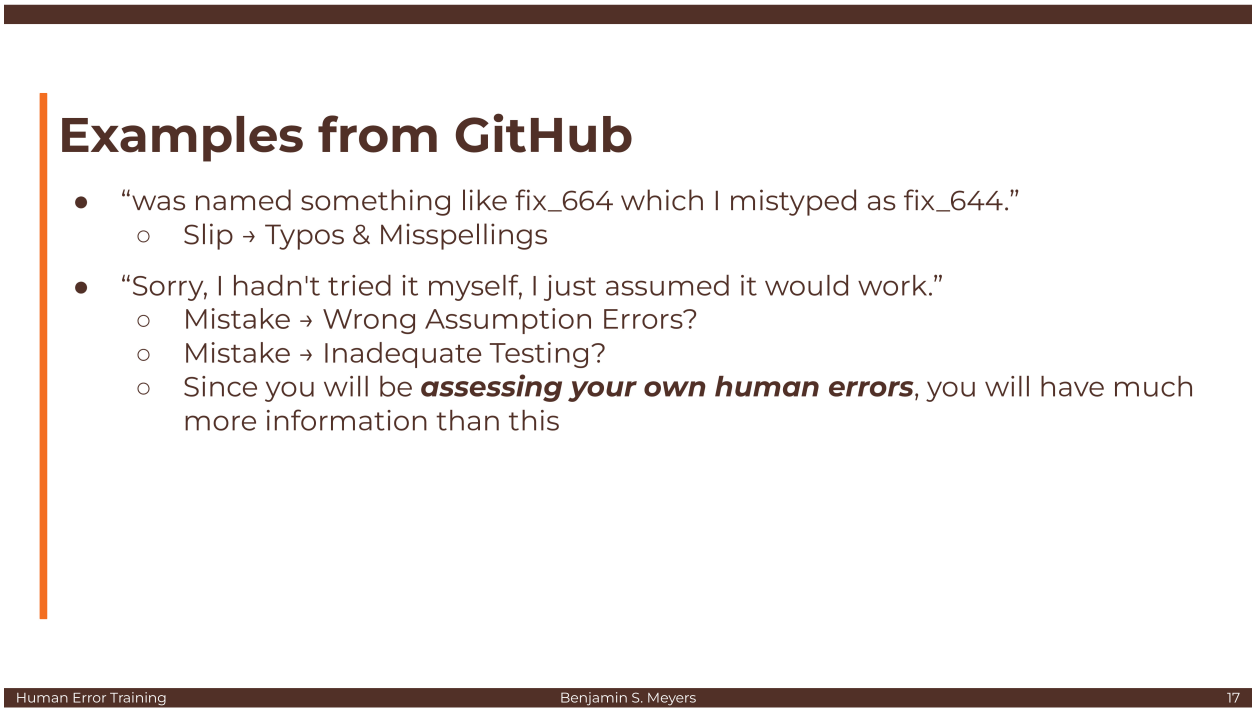
Examples from GitHub

- "in my test latexfile i forgot to write \pgfplotsset{compat=newest}"
  - Mistake → Incorrect/Insufficient Configuration
- "Whooooops! I made a mistake, I was actually using 2017a instead of 2017b on Windows 10."
  - Lapse → Working With Outdated Source Code
- "I made a mistake on using CPU clock time to calculate processing time."
  - Mistake → Code Logic Errors
- "Didn't read the ReadMe closely enough, sorry!"
  - Slip → Overlooking/Failure to Read Documentation

Human Error Training Benjamin S. Meyers 16

(p) Slide 16: Real-world examples of human errors in software engineering from GitHub

Figure D.1: Human Error Training Slides (Continued)

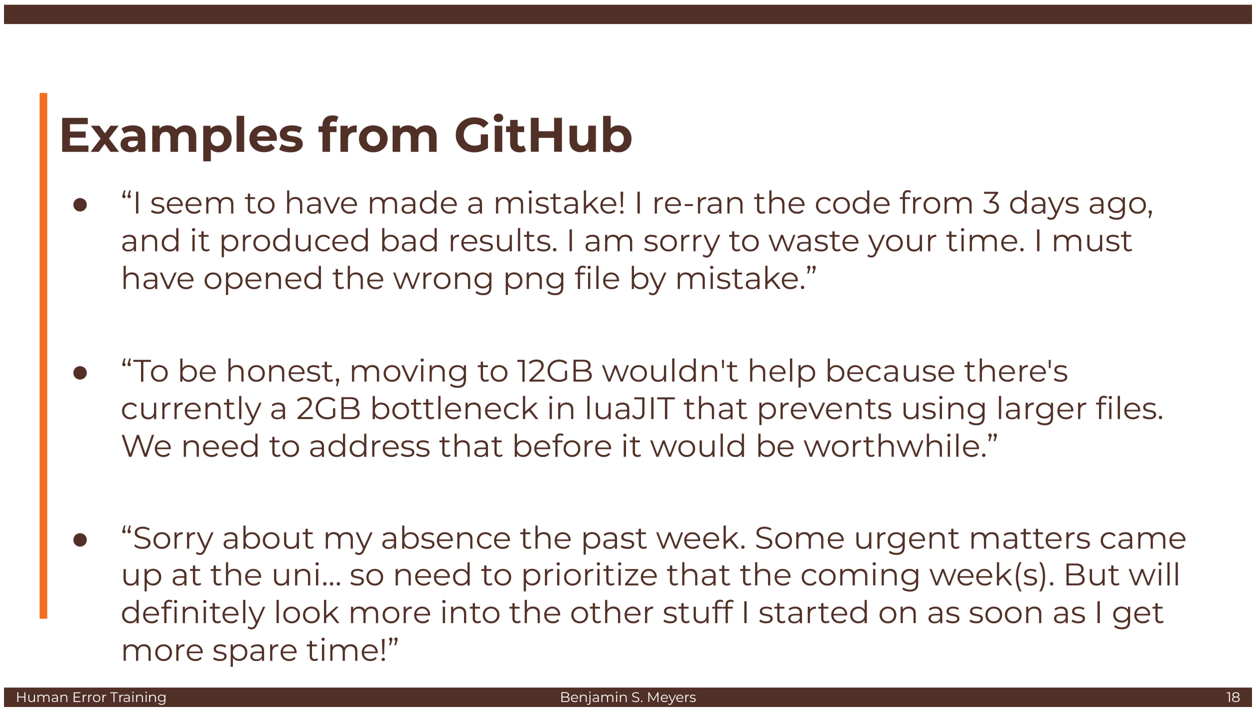


Examples from GitHub

- “was named something like fix_664 which I mistyped as fix_644.”
  - Slip → Typos & Misspellings
- “Sorry, I hadn't tried it myself, I just assumed it would work.”
  - Mistake → Wrong Assumption Errors?
  - Mistake → Inadequate Testing?
  - Since you will be ***assessing your own human errors***, you will have much more information than this

Human Error Training Benjamin S. Meyers 17

(q) Slide 17: Real-world examples of human errors in software engineering from GitHub



Examples from GitHub

- “I seem to have made a mistake! I re-ran the code from 3 days ago, and it produced bad results. I am sorry to waste your time. I must have opened the wrong png file by mistake.”
- “To be honest, moving to 12GB wouldn't help because there's currently a 2GB bottleneck in luaJIT that prevents using larger files. We need to address that before it would be worthwhile.”
- “Sorry about my absence the past week. Some urgent matters came up at the uni... so need to prioritize that the coming week(s). But will definitely look more into the other stuff I started on as soon as I get more spare time!”

Human Error Training Benjamin S. Meyers 18

(r) Slide 18: Discussion of more nuanced, open-ended human errors from GitHub

Figure D.1: Human Error Training Slides (Continued)

Appendix E

Open-Ended Final Survey Responses

This appendix includes anonymized responses to open-ended final survey questions (from Figure 5.3). Survey questions are repeated here for quick reference:

1. This research project...
   (a) was engaging.
   (b) had clear instructions.
   (c) enhanced my understanding of my own human errors.
   (d) involved a reasonable time commitment.
   (e) was a valuable learning experience.
   (f) enhanced my understanding of human errors in SE.
   (g) reinforced theoretical concepts related to human error.
   (h) involved a useful human error reporting form.
   (i) would benefit other SE students.
   (j) involved meaningful weekly discussions.
2. The taxonomy...
   (a) had clear descriptions and examples.
   (b) was simple to use for classifying my human errors.
   (c) was general enough to apply to all SE phases.
   (d) led to meaningful reflection on my human errors.
   (e) made it easy to organize and confront my human errors.
   (f) would be a beneficial tool for professional software engineers.
   (g) had categories that adequately described my human errors.
   (h) led to unambiguous classifications.
   (i) was confusing.
   (j) was overwhelming.
3. The taxonomy adequately covers potential human errors during...
   (a) software requirements engineering.
   (b) software design.
   (c) software implementation.
   (d) software testing.
   (e) software deployment.
   (f) software maintenance.
4. Please elaborate on your answers in the previous question.
5. Which type of human error (*i.e.* slips, lapses, or mistakes) do you believe has the most impact (*i.e.* consequence) on a software project and why?
6. What strategies, activities, processes, or tools would you recommend for identifying and/or preventing **slips**? Which strategy, activity, process, or tool would you say can **best** identify and/or prevent most **slips**? Please be specific.

7. What strategies, activities, processes, or tools would you recommend for identifying and/or preventing **lapses**? Which strategy, activity, process, or tool would you say can **best** identify and/or prevent most **lapses**? Please be specific.
8. What strategies, activities, processes, or tools would you recommend for identifying and/or preventing **mistakes**? Which strategy, activity, process, or tool would you say can **best** identify and/or prevent most **mistakes**? Please be specific.
9. Based on your experience during this research project, please indicate your level of agreement with the following statements.
   (a) Human errors in SE typically fall into one category.
   (b) Human errors in SE often span multiple categories.
   (c) One human error in SE can lead to others.
   (d) I can usually identify **slips** as they occur.
   (e) I can usually identify **lapses** as they occur.
   (f) I can usually identify **mistakes** as they occur.
10. If multiple developers experienced the same human error, do you believe they would place it into the same category?
    (a) Yes
    (b) No
    (c) Maybe
11. Please explain your answer to the previous question.
12. Do you believe that assessing human errors is a beneficial activity for software engineers?
    (a) Yes
    (b) No
    (c) Maybe
13. Please explain your answer to the previous question.
14. Please explain what you learned about your own human errors during this project.
15. Please describe whether or not you feel like you can better avoid your own human errors in the future.
16. Please describe any difficulties you had while learning about human errors (slips, lapses, and mistakes) or while using T.H.E.S.E. to classify your human errors.
17. Do you have any recommendations for improving T.H.E.S.E.?
18. If this research project were repeated (in a class, for example), what changes would you like to see?
19. Would you be interested in seeing your human errors in some aggregate form? Please describe what form would be useful for you.
20. Do you have any additional comments, concerns, or suggestions that you would like to share?

Table E.1: Open Ended Final Survey Responses

Q#	Respondent/Response
(4)	**A**: —
	B: While doing research, I would find that while it was difficult to identify human errors relating to software design because it is difficult to know if there was human error until after the design process.
	C: When doing a lot of design work, I didn't think the taxonomy would apply. Once Ben pointed out that it could be applied to the design phase as well, I realized that the taxonomy had a versatility I didn't realize previously. After this, I was able to apply the taxonomy to various areas of the SW development process.
	D: Requirements engineering can involve communications errors. Design can involve mistakes in understanding requirements, but depending on how clear the requirements are that's probably less likely. Implementation can have errors from all 3 categories (slips, lapses, mistakes). Testing can have errors in understanding what to test, and also slips and lapses, but can depend on the scope/understand-ability of the project. Deployment can involve slips and lapses for deployment procedures. Maintenance can have errors, but some errors in maintenance aren't covered (mistake in monitoring, recovery procedure errors, *etc.*).
	E: For the most part, it seemed like all the categories were covered. I just got into so odd edge cases during my requirement where something I did didn't really fit into a category - like forgetting to finish work I forgot to finish something I was developing and a diagram I was making. So I didn't forget to start it, just the wrapping up part.
(5)	**A**: —
	B: I would say mistakes have the most impact on a software project because slips and lapses have relatively simple solutions while mistakes lead to bigger changes in a software project.
	C: Lapses. I think lapses are most likely to result from carelessness or a disregard for the project. These types of errors can result in the downfall of a project faster than the other two because the developers, who are supposed to be the most attentive to the project, aren't satisfying this requirement.
	D: Mistakes. Harder to recognize and have more long-term consequences.
	E: Mistakes. In my experience, the majority of slips and lapses can be fixed pretty easily, like a typo can be fixed in less than a second after you find it. Even if you misspelled it multiple times, your IDE will normally have like a refactor feature where you can change all references to a misspelled variable at once. Mistakes usually mean that you have to take extra time to fix something that went wrong. You don't really realize your mistake until it's too late and you're dealing with the consequences of that.
(6)	**A**: —
	B: I think having a second person look over the code can help check for the minor things that were not caught by the developer. For coding slips, a compiler or debugger can help.
	C: The best way to prevent a slip is to maintain a strong understanding of what your own tasks are and what those tasks entail. This would involve discussions with your manager/supervisor/senior engineer to clear up and discrepancies in your understanding of the task.
	D: Compiler and IDEs, or CI for automated testing
	E: For me, most of my slips were caught by my IDE which checks my syntax and spelling as I go. The other types of slips tend to occur because you aren't paying attention to what you're really doing, so the best way to identify or prevent slips would be to just slow down and work intentionally. Software development tends to happen under pressure, like having a deadline or having to work on multiple things at once. I think the best way to identify / prevent would be to realign your focus before development - getting rid of distractions and taking time to understand fully what you are working on.
(7)	**A**: —
	B: Like slips, I think having someone else look over the code can help catch stuff that is missed. A project manager might be the best at identifying lapses with requirements.
	C: The best way to avoid lapses is to pay attention and take notes during briefings and stand ups. Without this practice, developers are more likely to entirely forget tasks or parts of tasks.
	D: Requirements testing, CI for automated testing, actively keeping track of backlog for current sprint.
	E: To prevent a lapse, you need to write stuff down and have it in a place where it can be easily accessed. A lot of my lapses were prevented by the notes I took during requirement gathering and development. It's hard to forget to implement a feature or fix a defect when you have it in your work queue in a program like JIRA or Trello. For stuff like forgetting an import statement, IDEs can tell you that pretty easily. Lapses like forgetting to save code or pull new code can be improved with good habit like if you make it habit to save your changes before leaving your computer or every so often, then you can help mitigate that.

continued on next page...

Table E.1: Open Ended Final Survey Responses (Continued)

Q#	Respondent/Response
(8)	**A**: —
	B: A way to prevent mistakes is to clearly define a process and following the process.
	C: Mistakes are the hardest to avoid because they are inevitable in the product development process. I think the best way to avoid mistakes would be to have a project manager + architect with strong understandings of the task at hand and adequate experience or domain knowledge towards guiding the developers to completing the project.
	D: Requirements testing and analysis. Verifying architecture with project owners. Considering process planning and design plan.
	E: To identity / prevent mistakes you need to have good planning practices. Doing even some form of planning can prevent a lot of mistakes from happening. The more time you take to plan, to establish a system of quality, the less mistakes you will encounter. Mostof these mistake categories sound like things we talked about in my [REDACTED COURSE NAME] classes. Like if you want quality testing, then follow the verification and validation model. Using the "7 Baisc Quality Tools": cause-and-effect diagram, checklist, pareto diagram, histogram, run chart, scatter plot, and control chart are all meant to get you to think about where issues in your process might occur, what are root causes, and what can you do to mitigate it. A lot of these mistakes take experience to identify and prevent - it's hard to plan around a problem you don't understand. If you don't have your own experience, then you can always ask someone else for their advice.
(11)	**A**: The labels and descriptions were generic enough to be understood by all, but specific enough to categorize specific problems generally experienced by all.
	B: I think their perception of the situation might lead to different placements of category.
	C: Yes, as long as the developers have an understanding of the three categories and the differences between them. I often found myself confusing lapse and slip towards the beginning because both involve not knowing information. The distinction, to me, is that a slip is forgetting information you once knew, and a lapse is not ever knowing a piece of information.
	D: Some errors can be a group effort (ex. a mistake in process planning is a mistake that the entire team made.)
	E: Some people just experience things differently. They might be thinking of the immediate error that just happened or they might think of the error that caused the error of the errors that resulted in it. If you misspell a word that prevents your program from running someone might call that a typo and someone else might call that a code logic error. Or maybe the word was misspelled in the requirements document so really it was a communication problem.
(13)	**A**: Very much. I can see how it's a tricky thing to study but it will definitely be worth it. Research like this could result to improved IDE functionality or even change the way languages are interpreted by humans.
	B: I think it is important to reflect on why we experience common errors and be proactive in preventing them rather than continuing a cycle of making similar mistakes and never knowing why.
	C: I think assessing human errors could be a very costly addition to a SW team's process, so I would only recommend the assessment of human errors as a part of the process for low priority projects and tasks. If a team used it though, the project manager would have a significantly better understanding of how individual developers work. I do think that this research has the potential to help us design new tools for IDE's and give project managers a new perspective on how the team is working.
	D: Depends on how much errors and how much time it takes to assess them. Overall useful but for some companies it might also be a waste of time.
	E: Understanding your issues is the first step to fixing them. If you log your errors and see you have are making the same mistake consistently, then that's a red flag that there's an issue you should probably fix.
(14)	**A**: I make silly mistakes that I get caught up on for too long
	B: I realized a lot of the human errors dealt with non-coding factors such as my stress levels or lack of knowledge.
	C: I realized that a lot of my mistakes occur when I'm trying to hit the ground running with a new tool. During the phase of exploration with a new tool, you're bound to assume things incorrectly, use bad syntax, or even misunderstand the concept behind the tool you're using. I realized I have lapses the least as I try to put effort into understanding my tasks and requirements before i complete them.
	D: Slips are typically made during development/implementation. Mistakes typically happen in pre-planning or design. Lapses can happen in both but are less likely overall.
	E: I learned that your brain autocorrects yourself all the time so a lot of slips / lapses happen without you even really registering it. Because of that, you can make the same mistake multiple times before you really pick up on it. I think that's also why I mostly reported mistakes, because they often had consequences that I had to take effort to deal with. There are some errors that I will never really be able to fully prevent- like I'm always going to misspell words, but because of that I know that I need to proofread.

continued on next page...

Table E.1: Open Ended Final Survey Responses (Continued)

Q#	Respondent/Response
(15)	**A**: Definitely will be able to navigate tricky errors better because I will understand what I'm doing wrong.
	B: I am unsure if I can avoid my own human errors, but I feel like I can better identify what went wrong when making human errors.
	C: The best way to avoid human errors is to think about what you're doing after you make a mistake. Trying to reflect on whether I knew a certain piece of necessary information or if I was properly equipped to handle a task is the best way to assess yourself.
	D: Depends on the error, as I get more experience I can experience the same errors in multiple projects, which would make me think that those errors are results of an anti-pattern.
	E: I feel like I can, having these definitions laid out in a list makes it a lot more digestible and kind of gives me a checklist of things to think about while I'm working on a project - is my testing strategy good enough? What's the best way to communicate something? Will this tool help me correct myself or is that something I need to plan for?
(16)	**A**: None.
	B: I did not have difficulty using T.H.E.S.E. to classify the human errors but I had difficulty catching my errors when they happened. A lot of my identifying was reflective.
	C: I felt like slips and lapses were confusing at first. Both terms remind me of "forget", and so whenever I talk about the two categories I have to think a little harder about what they specifically mean.
	D: N/A
	E: Sometimes I was just too busy to stop and think about my errors. Like I've made 5 typos in a row, I am not stopping to write down every single one. Sometimes it was difficult for me to decide if I should log the cause or effect of an error of the error I noticed. Also, I always forgot to check the document that got updated definitions so I mis-categorized a few times because of that, I always used the one in the form. The form is kind of long and it required thinking so for the most part I would just note my errors down and then report them later, but sometimes I would wait so long that I would forget what really happened so I had to be vague.
(17)	**A**: Ben knows from our discussions, just adding a few categories.
	B: I would consider process related errors.
	C: I don't have a good suggestion for fixing the slips/lapses confusion besides thinking of a different word to describe one or the other. I don't think that this confusion is enough to warrant larger changes to T.H.E.S.E.
	D: Maybe try pairing some of the errors to common anti-patterns in SE
	E: Maybe add in something for if an error was caused by another error. For me personally, I think a printed sheet of paper would have helped me. If I thought I was experiencing and error then I could just whip that out check real quick, looking stuff up on a computer is just annoying sometimes.
(18)	**A**: N/A
	B: I would want to try different projects (individual, group, implementation-related, design-related) to test all different aspects of software engineering.
	C: N/A
	D: An average metric for how critical some of these errors can affect a project.
	E: In a class, I think having a quota or goal for errors reported each week could help. I wasn't really sure how many I was expected to report so I wasn't always sure if I was doing as much as I was supposed to. I think it would interesting to have classmates interview each other for weekly reporting, it could give a student a different perspective on errors and increase their skill in identifying and reporting them.
(19)	**A**: It would be interesting+entertaining to see this info visualized on a web app. Maybe people could voluntarily add their info to a public web app that visualizes it.
	B: N/A
	C: I would be interested in seeing the human errors by frequency of category and perhaps with a list of descriptions in each category.
	D: Nah
	E: Just having the category and number or percentage reported would be enough for me to be interested. I would like to see the most common and least common, maybe which errors were reported together the most - like on the form where you could put if this error fit any other category. The most edge case errors would be interesting too.
(20)	**A**: N/A
	B: N/A
	C: N/A
	D: Cool research topic overall :D
	E: :)

Appendix F

Model Evaluation

In this appendix, we include model performance metrics for all 21 pretrained Sentence-BERT models evaluated in Chapter 5.3, both before and after improving T.H.E.S.E. category definitions. In the following tables, the highest value, second highest value, and lowest value for an experiment are coded in green, yellow, and red, respectively.

Table F.1: Multiclass Classification Results Before Improving T.H.E.S.E. Definitions

	With Apology Comments									**Without Apology Comments**								
	Slips			**Lapses**			**Mistakes**			**Slips**			**Lapses**			**Mistakes**		
Model	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**
	Without Natural Language Preprocessing																	
SB01	0.350	0.711	0.469	0.185	0.468	0.265	0.808	0.188	0.304	0.453	0.667	0.540	0.222	0.560	0.318	0.821	0.256	0.390
SB02	0.309	0.608	0.410	0.197	0.574	0.293	0.725	0.129	0.220	0.437	0.608	0.508	0.257	0.720	0.379	0.720	0.200	0.313
SB03	0.365	0.598	0.453	0.195	0.723	0.308	0.743	0.116	0.201	0.394	0.549	0.459	0.231	0.720	0.350	0.765	0.144	0.243
SB04	0.373	0.588	0.456	0.171	0.596	0.265	0.784	0.179	0.291	0.432	0.627	0.512	0.233	0.680	0.347	0.947	0.200	0.330
SB05	0.324	0.588	0.418	0.174	0.596	0.269	0.613	0.085	0.149	0.414	0.569	0.479	0.210	0.680	0.321	0.533	0.089	0.152
SB06	0.358	0.598	0.448	0.160	0.553	0.248	0.791	0.152	0.255	0.403	0.608	0.484	0.224	0.600	0.326	0.773	0.189	0.304
SB07	0.360	0.557	0.437	0.179	0.532	0.267	0.718	0.250	0.371	0.384	0.549	0.452	0.217	0.520	0.306	0.697	0.256	0.374
SB08	0.518	0.454	0.484	0.156	0.809	0.262	0.825	0.147	0.250	0.551	0.529	0.540	0.211	0.760	0.330	0.778	0.233	0.359
SB09	0.463	0.381	0.418	0.156	0.638	0.251	0.677	0.290	0.406	0.550	0.431	0.484	0.224	0.680	0.337	0.740	0.411	0.529
SB10	0.366	0.495	0.421	0.201	0.766	0.319	0.690	0.179	0.284	0.409	0.529	0.462	0.247	0.760	0.373	0.783	0.200	0.319
SB11	0.510	0.268	0.351	0.178	0.830	0.293	0.653	0.286	0.398	0.548	0.333	0.415	0.247	0.920	0.390	0.643	0.300	0.409
SB12	0.308	0.505	0.383	0.236	0.617	0.341	0.628	0.241	0.348	0.418	0.549	0.475	0.267	0.640	0.376	0.615	0.267	0.372
SB13	0.298	0.402	0.342	0.185	0.532	0.275	0.549	0.250	0.344	0.444	0.392	0.417	0.200	0.560	0.295	0.510	0.289	0.369
SB14	0.346	0.371	0.358	0.198	0.745	0.312	0.736	0.286	0.412	0.460	0.451	0.455	0.228	0.720	0.346	0.730	0.300	0.425
SB15	0.449	0.227	0.301	0.139	0.915	0.241	0.444	0.018	0.034	0.560	0.275	0.368	0.170	0.920	0.287	0.333	0.022	0.042
SB16	0.450	0.186	0.263	0.166	0.787	0.274	0.657	0.308	0.419	0.625	0.196	0.299	0.198	0.960	0.329	0.517	0.167	0.252
SB17	0.364	0.082	0.134	0.136	0.894	0.236	0.486	0.080	0.138	0.556	0.098	0.167	0.170	0.960	0.289	0.500	0.089	0.151
SB18	0.438	0.361	0.395	0.125	0.660	0.210	0.625	0.112	0.189	0.478	0.431	0.454	0.168	0.680	0.270	0.579	0.122	0.202
SB19	0.368	0.072	0.121	0.199	0.681	0.308	0.718	0.603	0.655	0.333	0.078	0.127	0.210	0.680	0.321	0.671	0.544	0.601
SB20	0.430	0.381	0.404	0.188	0.830	0.307	0.760	0.254	0.381	0.525	0.412	0.462	0.244	0.880	0.383	0.750	0.300	0.429
SB21	0.350	0.144	0.204	0.137	0.766	0.232	0.492	0.143	0.221	0.273	0.118	0.164	0.171	0.760	0.279	0.394	0.144	0.211
	With Natural Language Preprocessing																	
SB01	0.370	0.412	0.390	0.172	0.553	0.263	0.725	0.353	0.474	0.514	0.373	0.432	0.210	0.680	0.321	0.708	0.378	0.493
SB02	0.383	0.237	0.293	0.162	0.638	0.259	0.675	0.371	0.478	0.533	0.157	0.242	0.192	0.800	0.310	0.638	0.333	0.438
SB03	0.506	0.402	0.448	0.200	0.660	0.307	0.713	0.433	0.539	0.676	0.451	0.541	0.257	0.760	0.384	0.724	0.467	0.568
SB04	0.365	0.392	0.378	0.210	0.638	0.316	0.744	0.402	0.522	0.472	0.333	0.391	0.243	0.720	0.364	0.732	0.456	0.562
SB05	0.395	0.464	0.427	0.230	0.660	0.341	0.748	0.397	0.519	0.533	0.627	0.577	0.281	0.640	0.390	0.776	0.422	0.547
SB06	0.382	0.433	0.406	0.211	0.596	0.311	0.720	0.402	0.516	0.489	0.451	0.469	0.254	0.600	0.357	0.717	0.478	0.573
SB07	0.402	0.361	0.380	0.188	0.617	0.289	0.701	0.397	0.507	0.583	0.412	0.483	0.214	0.600	0.316	0.700	0.467	0.560
SB08	0.364	0.165	0.227	0.179	0.660	0.282	0.682	0.460	0.549	0.419	0.255	0.317	0.235	0.800	0.364	0.700	0.389	0.500
SB09	0.442	0.351	0.391	0.169	0.511	0.254	0.705	0.469	0.563	0.537	0.431	0.478	0.213	0.400	0.278	0.667	0.578	0.619
SB10	0.374	0.412	0.392	0.217	0.596	0.318	0.742	0.438	0.551	0.543	0.490	0.515	0.278	0.600	0.380	0.712	0.522	0.603
SB11	0.410	0.330	0.366	0.179	0.553	0.271	0.690	0.446	0.542	0.564	0.431	0.489	0.271	0.640	0.381	0.676	0.511	0.582
SB12	0.376	0.423	0.398	0.190	0.511	0.277	0.737	0.438	0.549	0.551	0.529	0.540	0.234	0.440	0.306	0.714	0.556	0.625
SB13	0.391	0.464	0.425	0.178	0.489	0.261	0.766	0.424	0.546	0.610	0.490	0.543	0.267	0.480	0.343	0.725	0.644	0.682
SB14	0.404	0.237	0.299	0.177	0.872	0.295	0.725	0.259	0.382	0.483	0.275	0.350	0.216	0.840	0.344	0.700	0.311	0.431
SB15	0.300	0.433	0.354	0.162	0.468	0.240	0.696	0.286	0.405	0.391	0.529	0.450	0.190	0.480	0.273	0.647	0.244	0.355
SB16	0.365	0.237	0.287	0.176	0.660	0.278	0.674	0.388	0.493	0.526	0.196	0.286	0.232	0.880	0.367	0.596	0.344	0.437
SB17	0.265	0.495	0.345	0.170	0.340	0.227	0.742	0.308	0.435	0.367	0.431	0.396	0.190	0.440	0.265	0.708	0.378	0.493
SB18	0.279	0.402	0.329	0.175	0.298	0.220	0.628	0.415	0.500	0.384	0.549	0.452	0.220	0.360	0.273	0.692	0.400	0.507
SB19	0.245	0.258	0.251	0.201	0.681	0.311	0.645	0.308	0.417	0.256	0.196	0.222	0.254	0.720	0.375	0.625	0.389	0.479
SB20	0.286	0.330	0.306	0.180	0.489	0.263	0.695	0.397	0.506	0.432	0.373	0.400	0.183	0.440	0.259	0.629	0.433	0.513
SB21	0.241	0.268	0.254	0.135	0.596	0.220	0.596	0.138	0.225	0.319	0.294	0.306	0.146	0.560	0.231	0.565	0.144	0.230

Table F.2: Binary Classification Results Before Improving T.H.E.S.E. Definitions

	With Apology Comments									**Without Apology Comments**								
	Slips			**Lapses**			**Mistakes**			**Slips**			**Lapses**			**Mistakes**		
Model	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**
	Without Natural Language Preprocessing																	
SB01	0.316	0.825	0.457	0.183	0.723	0.292	0.734	0.308	0.434	0.304	0.804	0.441	0.143	0.440	0.216	0.500	0.200	0.286
SB02	0.326	0.598	0.422	0.188	0.745	0.300	0.633	0.138	0.227	0.302	0.510	0.380	0.165	0.560	0.255	0.538	0.078	0.136
SB03	0.370	0.701	0.484	0.169	0.766	0.277	0.699	0.290	0.410	0.237	0.373	0.290	0.128	0.560	0.209	0.511	0.267	0.350
SB04	0.312	0.742	0.439	0.174	0.638	0.274	0.793	0.290	0.425	0.336	0.725	0.460	0.132	0.400	0.198	0.615	0.267	0.372
SB05	0.303	0.814	0.441	0.199	0.617	0.301	0.750	0.161	0.265	0.320	0.804	0.458	0.121	0.280	0.169	0.619	0.144	0.234
SB06	0.344	0.670	0.455	0.163	0.702	0.264	0.780	0.317	0.451	0.357	0.588	0.444	0.153	0.600	0.244	0.553	0.233	0.328
SB07	0.364	0.536	0.433	0.154	0.723	0.254	0.688	0.384	0.493	0.400	0.471	0.432	0.150	0.640	0.242	0.660	0.344	0.453
SB08	0.419	0.454	0.436	0.148	0.830	0.252	0.706	0.397	0.509	0.303	0.196	0.238	0.150	0.800	0.253	0.542	0.356	0.430
SB09	0.400	0.392	0.396	0.155	0.787	0.259	0.657	0.683	0.670	0.300	0.176	0.222	0.125	0.600	0.207	0.595	0.733	0.657
SB10	0.345	0.526	0.416	0.168	0.787	0.277	0.642	0.424	0.511	0.311	0.373	0.339	0.124	0.520	0.200	0.557	0.433	0.487
SB11	0.469	0.392	0.427	0.153	0.936	0.263	0.662	0.768	0.711	0.312	0.196	0.241	0.149	0.800	0.252	0.537	0.722	0.616
SB12	0.335	0.742	0.462	0.216	0.702	0.330	0.618	0.433	0.509	0.299	0.569	0.392	0.116	0.320	0.170	0.621	0.400	0.486
SB13	0.348	0.588	0.437	0.191	0.830	0.311	0.630	0.594	0.611	0.263	0.412	0.321	0.116	0.400	0.180	0.624	0.589	0.606
SB14	0.315	0.577	0.407	0.168	0.872	0.282	0.629	0.545	0.584	0.325	0.529	0.403	0.143	0.600	0.231	0.542	0.578	0.559
SB15	0.410	0.443	0.426	0.144	0.809	0.244	0.662	0.237	0.349	0.370	0.333	0.351	0.177	0.880	0.295	0.600	0.233	0.336
SB16	0.469	0.237	0.315	0.156	0.872	0.265	0.620	0.299	0.404	0.259	0.137	0.179	0.165	0.640	0.262	0.600	0.367	0.455
SB17	0.263	0.897	0.407	0.122	0.872	0.215	0.643	0.812	0.718	0.306	0.882	0.455	0.135	0.800	0.231	0.551	0.833	0.664
SB18	0.400	0.309	0.349	0.135	0.851	0.233	0.693	0.625	0.657	0.243	0.176	0.205	0.145	0.800	0.245	0.578	0.578	0.578
SB19	0.213	0.330	0.259	0.191	0.723	0.302	0.518	0.460	0.487	0.307	0.451	0.365	0.153	0.440	0.227	0.535	0.511	0.523
SB20	0.406	0.557	0.470	0.182	0.766	0.294	0.738	0.415	0.531	0.333	0.353	0.343	0.160	0.520	0.245	0.583	0.311	0.406
SB21	0.233	0.680	0.347	0.155	0.362	0.217	0.676	0.308	0.423	0.326	0.843	0.470	0.159	0.280	0.203	0.614	0.389	0.476
	With Natural Language Preprocessing																	
SB01	0.316	0.825	0.457	0.183	0.723	0.292	0.734	0.308	0.434	0.304	0.804	0.441	0.143	0.440	0.216	0.500	0.200	0.286
SB02	0.326	0.598	0.422	0.188	0.745	0.300	0.633	0.138	0.227	0.302	0.510	0.380	0.165	0.560	0.255	0.538	0.078	0.136
SB03	0.354	0.639	0.456	0.180	0.787	0.294	0.714	0.246	0.365	0.243	0.353	0.288	0.126	0.520	0.203	0.514	0.211	0.299
SB04	0.312	0.742	0.439	0.174	0.638	0.274	0.793	0.290	0.425	0.336	0.725	0.460	0.132	0.400	0.198	0.615	0.267	0.372
SB05	0.303	0.814	0.441	0.199	0.617	0.301	0.750	0.161	0.265	0.320	0.804	0.458	0.121	0.280	0.169	0.619	0.144	0.234
SB06	0.344	0.670	0.455	0.163	0.702	0.264	0.780	0.317	0.451	0.357	0.588	0.444	0.153	0.600	0.244	0.553	0.233	0.328
SB07	0.364	0.536	0.433	0.154	0.723	0.254	0.688	0.384	0.493	0.400	0.471	0.432	0.150	0.640	0.242	0.660	0.344	0.453
SB08	0.419	0.454	0.436	0.148	0.830	0.252	0.706	0.397	0.509	0.303	0.196	0.238	0.150	0.800	0.253	0.542	0.356	0.430
SB09	0.400	0.392	0.396	0.155	0.787	0.259	0.657	0.683	0.670	0.300	0.176	0.222	0.125	0.600	0.207	0.595	0.733	0.657
SB10	0.345	0.526	0.416	0.168	0.787	0.277	0.642	0.424	0.511	0.311	0.373	0.339	0.124	0.520	0.200	0.557	0.433	0.487
SB11	0.469	0.392	0.427	0.153	0.936	0.263	0.662	0.768	0.711	0.312	0.196	0.241	0.149	0.800	0.252	0.537	0.722	0.616
SB12	0.335	0.742	0.462	0.216	0.702	0.330	0.618	0.433	0.509	0.299	0.569	0.392	0.116	0.320	0.170	0.621	0.400	0.486
SB13	0.348	0.588	0.437	0.191	0.830	0.311	0.630	0.594	0.611	0.263	0.412	0.321	0.116	0.400	0.180	0.624	0.589	0.606
SB14	0.315	0.577	0.407	0.168	0.872	0.282	0.629	0.545	0.584	0.325	0.529	0.403	0.143	0.600	0.231	0.542	0.578	0.559
SB15	0.410	0.443	0.426	0.144	0.809	0.244	0.662	0.237	0.349	0.370	0.333	0.351	0.177	0.880	0.295	0.600	0.233	0.336
SB16	0.469	0.237	0.315	0.156	0.872	0.265	0.620	0.299	0.404	0.259	0.137	0.179	0.165	0.640	0.262	0.600	0.367	0.455
SB17	0.263	0.897	0.407	0.122	0.872	0.215	0.643	0.812	0.718	0.306	0.882	0.455	0.135	0.800	0.231	0.551	0.833	0.664
SB18	0.400	0.309	0.349	0.135	0.851	0.233	0.693	0.625	0.657	0.243	0.176	0.205	0.145	0.800	0.245	0.578	0.578	0.578
SB19	0.213	0.330	0.259	0.191	0.723	0.302	0.518	0.460	0.487	0.307	0.451	0.365	0.153	0.440	0.227	0.535	0.511	0.523
SB20	0.406	0.557	0.470	0.182	0.766	0.294	0.738	0.415	0.531	0.333	0.353	0.343	0.160	0.520	0.245	0.583	0.311	0.406
SB21	0.233	0.680	0.347	0.155	0.362	0.217	0.676	0.308	0.423	0.326	0.843	0.470	0.159	0.280	0.203	0.614	0.389	0.476

Table F.3: Multiclass Classification Results After Improving T.H.E.S.E. Definitions

	With Apology Comments									**Without Apology Comments**								
	Slips			**Lapses**			**Mistakes**			**Slips**			**Lapses**			**Mistakes**		
Model	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**
	Without Natural Language Preprocessing																	
SB01	0.306	0.454	0.365	0.183	0.426	0.256	0.626	0.321	0.425	0.404	0.451	0.426	0.211	0.480	0.293	0.558	0.322	0.408
SB02	0.368	0.505	0.426	0.162	0.596	0.255	0.661	0.183	0.287	0.562	0.529	0.545	0.207	0.680	0.318	0.667	0.267	0.381
SB03	0.460	0.474	0.467	0.174	0.723	0.281	0.726	0.237	0.357	0.524	0.431	0.473	0.188	0.640	0.291	0.692	0.300	0.419
SB04	0.383	0.588	0.463	0.158	0.532	0.244	0.754	0.205	0.323	0.493	0.647	0.559	0.192	0.560	0.286	0.846	0.244	0.379
SB05	0.318	0.577	0.410	0.167	0.468	0.246	0.617	0.165	0.261	0.409	0.529	0.462	0.192	0.560	0.286	0.593	0.178	0.274
SB06	0.360	0.639	0.461	0.121	0.340	0.179	0.672	0.192	0.299	0.461	0.686	0.551	0.161	0.400	0.230	0.679	0.211	0.322
SB07	0.429	0.433	0.431	0.164	0.660	0.263	0.654	0.237	0.348	0.500	0.490	0.495	0.187	0.560	0.280	0.634	0.289	0.397
SB08	0.489	0.443	0.465	0.159	0.766	0.264	0.796	0.192	0.309	0.540	0.529	0.535	0.210	0.680	0.321	0.771	0.300	0.432
SB09	0.447	0.392	0.418	0.140	0.617	0.228	0.566	0.192	0.287	0.500	0.529	0.514	0.182	0.560	0.275	0.629	0.244	0.352
SB10	0.394	0.515	0.446	0.184	0.681	0.290	0.657	0.196	0.302	0.469	0.588	0.522	0.225	0.640	0.333	0.677	0.233	0.347
SB11	0.500	0.227	0.312	0.154	0.809	0.259	0.590	0.205	0.305	0.556	0.294	0.385	0.216	0.880	0.346	0.622	0.256	0.362
SB12	0.317	0.526	0.395	0.140	0.319	0.195	0.600	0.268	0.370	0.406	0.549	0.467	0.212	0.440	0.286	0.622	0.311	0.415
SB13	0.338	0.474	0.395	0.150	0.404	0.218	0.571	0.268	0.365	0.451	0.451	0.451	0.188	0.520	0.277	0.543	0.278	0.368
SB14	0.291	0.309	0.300	0.180	0.809	0.295	0.741	0.179	0.288	0.400	0.275	0.326	0.208	0.840	0.333	0.700	0.233	0.350
SB15	0.294	0.361	0.324	0.141	0.617	0.229	0.488	0.094	0.157	0.288	0.294	0.291	0.172	0.680	0.274	0.333	0.056	0.095
SB16	0.545	0.124	0.202	0.166	0.872	0.279	0.576	0.254	0.353	0.786	0.216	0.338	0.202	0.960	0.333	0.515	0.189	0.276
SB17	0.194	0.268	0.225	0.105	0.170	0.130	0.532	0.375	0.440	0.217	0.255	0.234	0.200	0.200	0.200	0.481	0.433	0.456
SB18	0.397	0.278	0.327	0.144	0.787	0.243	0.674	0.129	0.217	0.500	0.314	0.386	0.175	0.800	0.288	0.600	0.133	0.218
SB19	0.347	0.268	0.302	0.076	0.213	0.112	0.506	0.366	0.425	0.372	0.314	0.340	0.133	0.240	0.171	0.449	0.389	0.417
SB20	0.374	0.381	0.378	0.187	0.787	0.302	0.535	0.170	0.258	0.525	0.412	0.462	0.278	0.880	0.423	0.574	0.300	0.394
SB21	0.262	0.175	0.210	0.140	0.638	0.230	0.494	0.196	0.281	0.269	0.137	0.182	0.160	0.600	0.252	0.457	0.233	0.309
	With Natural Language Preprocessing																	
SB01	0.281	0.443	0.344	0.198	0.447	0.275	0.560	0.272	0.366	0.375	0.412	0.393	0.241	0.520	0.329	0.518	0.322	0.397
SB02	0.300	0.845	0.443	0.208	0.319	0.252	0.739	0.076	0.138	0.414	0.804	0.547	0.269	0.560	0.364	0.867	0.144	0.248
SB03	0.434	0.577	0.496	0.175	0.660	0.277	0.742	0.205	0.322	0.500	0.608	0.549	0.169	0.480	0.250	0.758	0.278	0.407
SB04	0.411	0.598	0.487	0.167	0.617	0.262	0.774	0.183	0.296	0.500	0.647	0.564	0.215	0.680	0.327	0.905	0.211	0.342
SB05	0.318	0.660	0.430	0.163	0.426	0.235	0.636	0.125	0.209	0.405	0.627	0.492	0.200	0.520	0.289	0.636	0.156	0.250
SB06	0.359	0.670	0.468	0.126	0.362	0.187	0.731	0.170	0.275	0.453	0.667	0.540	0.167	0.400	0.235	0.710	0.244	0.364
SB07	0.406	0.402	0.404	0.170	0.745	0.277	0.621	0.183	0.283	0.480	0.471	0.475	0.198	0.640	0.302	0.571	0.222	0.320
SB08	0.473	0.443	0.457	0.153	0.787	0.256	0.771	0.121	0.208	0.500	0.529	0.514	0.205	0.720	0.319	0.750	0.200	0.316
SB09	0.452	0.392	0.420	0.152	0.723	0.252	0.623	0.170	0.267	0.510	0.510	0.510	0.200	0.640	0.305	0.743	0.289	0.416
SB10	0.413	0.464	0.437	0.155	0.660	0.251	0.627	0.165	0.261	0.441	0.510	0.473	0.188	0.600	0.286	0.630	0.189	0.291
SB11	0.511	0.237	0.324	0.157	0.830	0.264	0.600	0.201	0.301	0.593	0.314	0.410	0.212	0.880	0.341	0.657	0.256	0.368
SB12	0.349	0.526	0.420	0.124	0.277	0.171	0.624	0.326	0.428	0.481	0.510	0.495	0.192	0.400	0.260	0.633	0.422	0.507
SB13	0.337	0.309	0.323	0.146	0.319	0.200	0.602	0.473	0.530	0.545	0.353	0.429	0.218	0.480	0.300	0.603	0.522	0.560
SB14	0.274	0.330	0.299	0.177	0.809	0.290	0.722	0.116	0.200	0.341	0.275	0.304	0.208	0.840	0.333	0.708	0.189	0.298
SB15	0.269	0.258	0.263	0.139	0.574	0.224	0.506	0.183	0.269	0.289	0.216	0.247	0.177	0.680	0.281	0.406	0.144	0.213
SB16	0.529	0.186	0.275	0.173	0.915	0.291	0.600	0.228	0.330	0.812	0.255	0.388	0.200	0.960	0.331	0.533	0.178	0.267
SB17	0.302	0.134	0.186	0.122	0.319	0.176	0.569	0.513	0.540	0.364	0.157	0.219	0.204	0.400	0.270	0.526	0.556	0.541
SB18	0.396	0.196	0.262	0.133	0.745	0.226	0.632	0.161	0.256	0.524	0.216	0.306	0.168	0.760	0.275	0.594	0.211	0.311
SB19	0.190	0.299	0.232	0.125	0.106	0.115	0.509	0.397	0.446	0.227	0.294	0.256	0.118	0.080	0.095	0.458	0.422	0.439
SB20	0.437	0.464	0.450	0.180	0.787	0.294	0.633	0.170	0.268	0.622	0.549	0.583	0.253	0.840	0.389	0.684	0.289	0.406
SB21	0.224	0.639	0.332	0.158	0.128	0.141	0.453	0.107	0.173	0.248	0.549	0.341	0.208	0.200	0.204	0.414	0.133	0.202

Table F.4: Binary Classification Results After Improving T.H.E.S.E. Definitions

	With Apology Comments									Without Apology Comments								
	Slips			Lapses			Mistakes			Slips			Lapses			Mistakes		
Model	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**	**Pr.**	**Re.**	**F1**
	Without Natural Language Preprocessing																	
SB01	0.318	0.794	0.454	0.175	0.468	0.254	0.640	0.388	0.483	0.322	0.765	0.453	0.222	0.400	0.286	0.578	0.411	0.481
SB02	0.335	0.608	0.432	0.161	0.660	0.258	0.663	0.263	0.377	0.301	0.490	0.373	0.165	0.560	0.255	0.371	0.144	0.208
SB03	0.401	0.588	0.477	0.165	0.787	0.273	0.719	0.308	0.431	0.273	0.353	0.308	0.155	0.640	0.250	0.571	0.267	0.364
SB04	0.342	0.691	0.457	0.163	0.596	0.256	0.714	0.312	0.435	0.355	0.647	0.458	0.178	0.520	0.265	0.622	0.311	0.415
SB05	0.321	0.722	0.444	0.169	0.532	0.256	0.643	0.241	0.351	0.278	0.588	0.377	0.172	0.400	0.241	0.459	0.189	0.268
SB06	0.340	0.732	0.464	0.126	0.426	0.194	0.750	0.388	0.512	0.362	0.667	0.469	0.181	0.520	0.268	0.537	0.322	0.403
SB07	0.395	0.619	0.482	0.145	0.681	0.240	0.672	0.411	0.510	0.317	0.392	0.351	0.163	0.680	0.264	0.588	0.444	0.506
SB08	0.417	0.515	0.461	0.145	0.766	0.244	0.688	0.442	0.538	0.368	0.275	0.315	0.164	0.840	0.275	0.514	0.400	0.450
SB09	0.422	0.474	0.447	0.147	0.809	0.248	0.645	0.665	0.655	0.212	0.137	0.167	0.143	0.760	0.241	0.590	0.800	0.679
SB10	0.366	0.649	0.468	0.189	0.787	0.305	0.625	0.357	0.455	0.303	0.392	0.342	0.150	0.600	0.240	0.567	0.378	0.453
SB11	0.485	0.340	0.400	0.147	0.936	0.254	0.663	0.737	0.698	0.286	0.118	0.167	0.152	0.880	0.259	0.533	0.711	0.610
SB12	0.322	0.763	0.453	0.130	0.383	0.195	0.645	0.446	0.528	0.315	0.667	0.428	0.103	0.240	0.145	0.587	0.411	0.484
SB13	0.349	0.773	0.481	0.170	0.553	0.260	0.644	0.549	0.593	0.321	0.686	0.438	0.105	0.240	0.146	0.584	0.500	0.539
SB14	0.289	0.340	0.313	0.159	0.851	0.268	0.675	0.500	0.574	0.237	0.275	0.255	0.142	0.600	0.229	0.518	0.322	0.397
SB15	0.307	0.443	0.363	0.145	0.702	0.240	0.629	0.326	0.429	0.284	0.373	0.322	0.162	0.640	0.258	0.538	0.311	0.394
SB16	0.542	0.268	0.359	0.140	0.957	0.245	0.643	0.714	0.677	0.450	0.176	0.254	0.151	0.880	0.257	0.563	0.789	0.657
SB17	0.273	0.619	0.379	0.115	0.362	0.174	0.564	0.491	0.525	0.283	0.510	0.364	0.122	0.360	0.182	0.547	0.522	0.534
SB18	0.362	0.299	0.328	0.132	0.809	0.227	0.624	0.473	0.538	0.184	0.137	0.157	0.141	0.720	0.235	0.538	0.467	0.500
SB19	0.371	0.268	0.311	0.143	0.830	0.244	0.523	0.402	0.455	0.357	0.196	0.253	0.143	0.720	0.238	0.560	0.467	0.509
SB20	0.388	0.557	0.458	0.170	0.830	0.283	0.612	0.330	0.429	0.258	0.314	0.283	0.183	0.760	0.295	0.442	0.211	0.286
SB21	0.291	0.258	0.273	0.145	0.872	0.249	0.510	0.228	0.315	0.318	0.275	0.295	0.148	0.720	0.245	0.649	0.267	0.378
	With Natural Language Preprocessing																	
SB01	0.320	0.814	0.459	0.190	0.489	0.274	0.583	0.344	0.433	0.314	0.725	0.438	0.208	0.400	0.274	0.542	0.356	0.430
SB02	0.292	0.794	0.427	0.182	0.340	0.237	0.727	0.071	0.130	0.304	0.804	0.441	0.222	0.240	0.231	0.500	0.033	0.062
SB03	0.392	0.670	0.494	0.177	0.745	0.286	0.735	0.272	0.397	0.254	0.353	0.295	0.141	0.560	0.226	0.457	0.178	0.256
SB04	0.379	0.660	0.481	0.156	0.660	0.252	0.738	0.353	0.477	0.338	0.510	0.406	0.135	0.480	0.211	0.642	0.378	0.476
SB05	0.328	0.784	0.462	0.158	0.447	0.233	0.667	0.188	0.293	0.288	0.627	0.395	0.179	0.400	0.247	0.484	0.167	0.248
SB06	0.333	0.763	0.464	0.130	0.404	0.197	0.713	0.321	0.443	0.340	0.686	0.455	0.175	0.440	0.250	0.535	0.256	0.346
SB07	0.383	0.474	0.424	0.148	0.745	0.246	0.655	0.348	0.455	0.360	0.353	0.356	0.152	0.680	0.248	0.642	0.378	0.476
SB08	0.420	0.485	0.450	0.148	0.809	0.251	0.689	0.366	0.478	0.333	0.216	0.262	0.158	0.840	0.266	0.526	0.333	0.408
SB09	0.377	0.443	0.408	0.154	0.830	0.259	0.654	0.549	0.597	0.267	0.157	0.198	0.154	0.840	0.261	0.567	0.611	0.588
SB10	0.392	0.577	0.467	0.156	0.745	0.257	0.625	0.379	0.472	0.327	0.333	0.330	0.149	0.680	0.245	0.530	0.389	0.449
SB11	0.470	0.320	0.380	0.142	0.915	0.246	0.664	0.732	0.696	0.304	0.137	0.189	0.154	0.880	0.262	0.526	0.678	0.592
SB12	0.328	0.773	0.460	0.129	0.383	0.194	0.654	0.531	0.586	0.315	0.667	0.428	0.121	0.280	0.169	0.614	0.478	0.537
SB13	0.338	0.701	0.456	0.156	0.553	0.243	0.642	0.737	0.686	0.317	0.627	0.421	0.092	0.240	0.133	0.548	0.700	0.615
SB14	0.281	0.351	0.312	0.162	0.851	0.272	0.570	0.598	0.584	0.230	0.275	0.250	0.143	0.600	0.231	0.520	0.589	0.552
SB15	0.333	0.443	0.381	0.151	0.766	0.252	0.608	0.464	0.527	0.338	0.431	0.379	0.158	0.640	0.254	0.590	0.511	0.548
SB16	0.493	0.371	0.424	0.152	0.936	0.262	0.641	0.558	0.597	0.400	0.314	0.352	0.172	0.840	0.286	0.576	0.589	0.582
SB17	0.374	0.351	0.362	0.123	0.723	0.210	0.603	0.759	0.672	0.323	0.196	0.244	0.148	0.800	0.250	0.519	0.756	0.615
SB18	0.390	0.237	0.295	0.123	0.809	0.213	0.623	0.701	0.660	0.250	0.137	0.177	0.152	0.840	0.258	0.495	0.589	0.538
SB19	0.196	0.309	0.240	0.164	0.745	0.268	0.519	0.438	0.475	0.278	0.392	0.325	0.152	0.560	0.239	0.524	0.478	0.500
SB20	0.447	0.474	0.460	0.167	0.830	0.279	0.655	0.330	0.439	0.302	0.255	0.277	0.179	0.760	0.290	0.455	0.222	0.299
SB21	0.226	0.649	0.335	0.135	0.255	0.176	0.468	0.129	0.203	0.333	0.882	0.484	0.194	0.240	0.214	0.625	0.167	0.263

Table F.5: Summary of Human Error Classifications After Improving T.H.E.S.E. Definitions

		Assigned					**Assigned**		
	Total	**Slip**	**Lapse**	**Mistake**		**Total**	**Slip**	**Lapse**	**Mistake**
Slips					***Mistakes***				
S01	25	15	3	7	**M01**	41	7	10	24
S02	6	6	0	0	**M02**	2	0	2	0
S03	32	18	8	6	**M03**	32	7	11	16
S04	2	1	1	0	**M04**	16	4	10	2
S05	18	6	8	4	**M05**	19	4	11	4
S06	3	0	2	1	**M06**	7	3	2	2
S07	2	0	1	1	**M07**	39	12	25	2
S08	9	3	3	3	**M08**	3	1	2	0
Lapses					**M09**	14	3	8	3
L01	4	3	1	0	**M10**	8	2	2	4
L02	2	1	0	1	**M11**	3	2	1	0
L03	6	2	2	2	**M12**	17	3	8	6
L04	17	9	3	5	**M13**	6	1	4	1
L05	3	0	2	1	**M14**	16	2	11	3
L06	6	5	1	0	**M15**	1	1	0	0
L07	4	2	1	1	***Other***				
L08	5	4	1	0		—	—	—	—

NOTE: See Table 5.10 for this summary before improving T.H.E.S.E. descriptions.

Appendix G

Extended T.H.E.S.E. Descriptions

This appendix includes the extended T.H.E.S.E. category descriptions as described in Section 5.3.2.4. Note that the CWE identifiers shown in Table G.1 are provided for reference only; they were not included in the classification experiment discussed in Section 5.3.2.4.

Table G.1: Extended T.H.E.S.E. Descriptions from CWE [235] and HET [12]

	Extended Description
Slips	
S02	**CWE-1114**: Inappropriate Whitespace Style. The source code contains whitespace that is inconsistent across the code or does not follow expected standards for the product.
S03	**CWE-477**: Use of Obsolete Function. The code uses deprecated or obsolete functions, which suggests that the code has not been actively reviewed or maintained. **CWE-1068**: Inconsistency Between Implementation and Documented Design. The implementation of the product is not consistent with the design as described within the relevant documentation.
S07	**CWE-1041**: Use of Redundant Code. The product has multiple functions, methods, procedures, macros, *etc.* that contain the same code.
S08	**Clerical Errors**. Result from carelessness while performing mechanical transcriptions from one format or from one medium to another. Requirement examples include carelessness while documenting specifications from elicited user needs. **Term Substitution Errors**. After introducing a term correctly, the requirement author substitutes a different term that refers to a different concept.
Lapses	
L01	**CWE-820**: Missing Synchronization. The product utilizes a shared resource in a concurrent manner but does not attempt to synchronize access to the resource. **CWE-325**: Missing Cryptographic Step. The product does not implement a required step in a cryptographic algorithm, resulting in weaker encryption than advertised by the algorithm.
L03	**CWE-117**: Improper Output Neutralization for Logs. The product does not neutralize or incorrectly neutralizes output that is written to logs. **CWE-489**: Active Debug Code. The product is deployed to unauthorized actors with debugging code still enabled or active, which can create unintended entry points or expose sensitive information. **CWE-561**: Dead Code. The product contains dead code, which can never be executed. **CWE-1164**: Irrelevant Code. The product contains code that is not essential for execution, *i.e.* makes no state changes and has no side effects that alter data or control flow, such that removal of the code would have no impact to functionality or correctness.
L08	**Loss of Information from Stakeholders**. Result from a requirement author forgetting, discarding, or failing to store information or documents provided by stakeholders, *e.g.* some important user need. **Accidentally Overlooking Requirements**. Occur when the stakeholders who are the source of requirements assume that some requirements are obvious and fail to verbalize them. **Multiple Terms for the Same Concept**. Occur when requirement authors fail to realize they have already defined a term for a concept and so introduce a new term at a later time.

continued on next page...

Table G.1: Extended T.H.E.S.E. Descriptions from CWE [235] and HET [12] (Continued)

	Extended Description
Mistakes	
M01	**CWE-280**: Improper Handling of Insufficient Permissions or Privileges. The product does not handle or incorrectly handles when it has insufficient privileges to access resources or functionality as specified by their permissions. This may cause it to follow unexpected code paths that may leave the product in an invalid state. **CWE-478**: Missing Default Case in Multiple Condition Expression. The code does not have a default case in an expression with multiple conditions, such as a switch statement. **CWE-628**: Function Call with Incorrectly Specified Arguments. The product calls a function, procedure, or routine with arguments that are not correctly specified, leading to always-incorrect behavior and resultant weaknesses. **CWE-480**: Use of Incorrect Operator. The product accidentally uses the wrong operator, which changes the logic in security-relevant ways. **CWE-484**: Omitted Break Statement in Switch. The product omits a break statement within a switch or similar construct, causing code associated with multiple conditions to execute. This can cause problems when the programmer only intended to execute code associated with one condition. **CWE-783**: Operator Precedence Logic Error. The product uses an expression in which operator precedence causes incorrect logic to be used. **CWE-835**: Loop with Unreachable Exit Condition ('Infinite Loop'). The product contains an iteration or loop with an exit condition that cannot be reached, *i.e.* , an infinite loop. **CWE-1025**: Comparison Using Wrong Factors. The code performs a comparison between two entities, but the comparison examines the wrong factors or characteristics of the entities, which can lead to incorrect results and resultant weaknesses. **CWE-367**: Time-of-check Time-of-use (TOCTOU) Race Condition. The product checks the state of a resource before using that resource, but the resource's state can change between the check and the use in a way that invalidates the results of the check. This can cause the product to perform invalid actions when the resource is in an unexpected state. **CWE-128**: Wrap-around Error. Wrap around errors occur whenever a value is incremented past the maximum value for its type and therefore 'wraps around' to a very small, negative, or undefined value. **CWE-369**: Divide by Zero. The product divides a value by zero.
M03	**CWE-20**: Improper Input Validation. The product receives input or data, but it does not validate or incorrectly validates that the input has the properties that are required to process the data safely and correctly.
M06	**CWE-308**: Use of Single-factor Authentication. The use of single-factor authentication can lead to unnecessary risk of compromise when compared with the benefits of a dual-factor authentication scheme. **CWE-603**: Use of Client-Side Authentication. A client/server product performs authentication within client code but not in server code, allowing server-side authentication to be bypassed via a modified client that omits the authentication check. **CWE-547**: Use of Hard-coded, Security-relevant Constants. The product uses hard-coded constants instead of symbolic names for security-critical values, which increases the likelihood of mistakes during code maintenance or security policy change. **CWE-1104**: Use of Unmaintained Third Party Components. The product relies on third-party components that are not actively supported or maintained by the original developer or a trusted proxy for the original developer. **CWE-798**: Use of Hard-coded Credentials. The product contains hard-coded credentials, such as a password or cryptographic key, which it uses for its own inbound authentication, outbound communication to external components, or encryption of internal data.

continued on next page...

Table G.1: Extended T.H.E.S.E. Descriptions from CWE [235] and HET [12] (Continued)

	Extended Description
Mistakes	
M08	**CWE-754**: Improper Check for Unusual or Exceptional Conditions. The product does not check or incorrectly checks for unusual or exceptional conditions that are not expected to occur frequently during day to day operation of the product.
M09	**CWE-266**: Incorrect Privilege Assignment. A product incorrectly assigns a privilege to a particular actor, creating an unintended sphere of control for that actor. **CWE-1220**: Insufficient Granularity of Access Control. The product implements access controls via a policy or other feature with the intention to disable or restrict accesses (reads and/or writes) to assets in a system from untrusted agents. However, implemented access controls lack required granularity, which renders the control policy too broad because it allows accesses from unauthorized agents to the security-sensitive assets. **CWE-1392**: Use of Default Credentials. The product uses default credentials (such as passwords or cryptographic keys) for potentially critical functionality.
M10	**CWE-1099**: Inconsistent Naming Conventions for Identifiers. The product's code, documentation, or other artifacts do not consistently use the same naming conventions for variables, callables, groups of related callables, I/O capabilities, data types, file names, or similar types of elements. **CWE-1109**: Use of Same Variable for Multiple Purposes. The code contains a callable, block, or other code element in which the same variable is used to control more than one unique task or store more than one instance of data. **CWE-1074**: Class with Excessively Deep Inheritance. A class has an inheritance level that is too high, *i.e.* , it has a large number of parent classes. **CWE-1080**: Source Code File with Excessive Number of Lines of Code. A source code file has too many lines of code. **CWE-1124**: Excessively Deep Nesting. The code contains a callable or other code grouping in which the nesting / branching is too deep.
M11	**CWE-176**: Improper Handling of Unicode Encoding. The product does not properly handle when an input contains Unicode encoding. **CWE-173**: Improper Handling of Alternate Encoding. The product does not properly handle when an input uses an alternate encoding that is valid for the control sphere to which the input is being sent.

continued on next page...

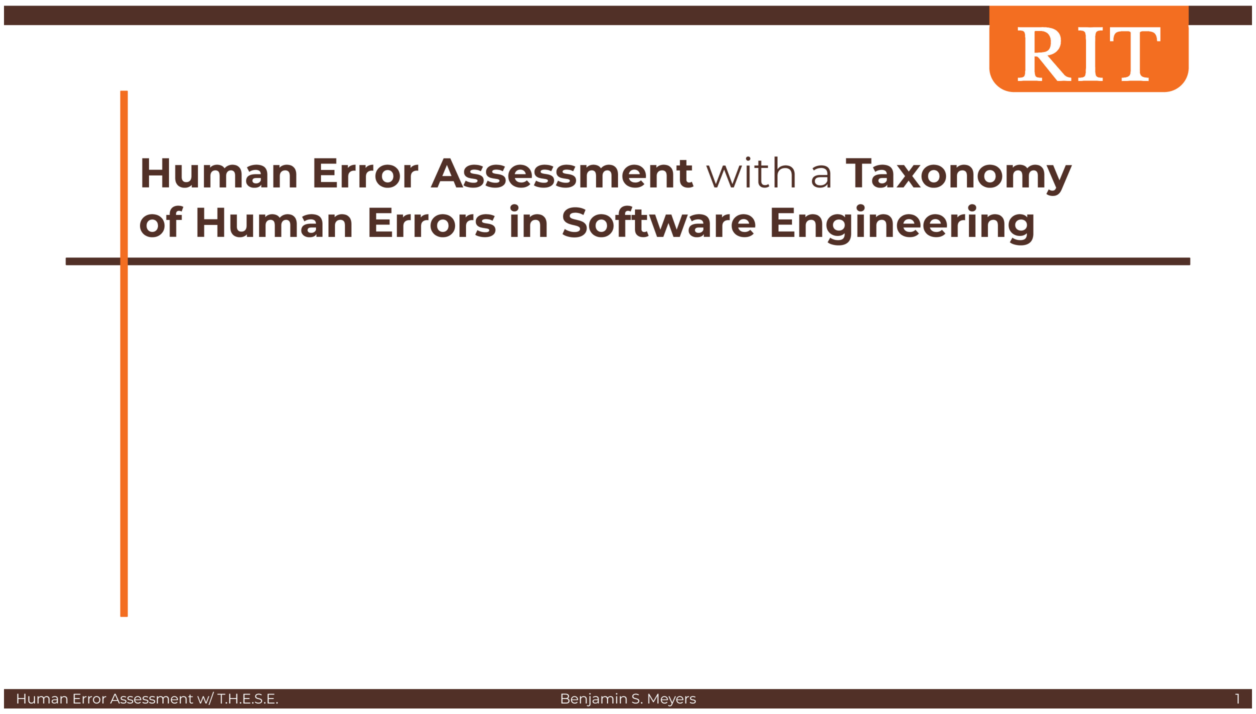
Table G.1: Extended T.H.E.S.E. Descriptions from CWE [235] and HET [12] (Continued)

	Extended Description
Mistakes	
M15	**Application**. Arise from a misunderstanding of the application or problem domain or a misunderstanding of some aspect of overall system functionality. **Environment**. Result from lack of knowledge about the available infrastructure (*e.g.* , tools, templates) that supports the elicitation, understanding, or documentation of software requirements. **Solution Choice**. These errors occur in the process of finding a solution for a stated and well-understood problem. If RE analysts do not understand the correct use of problem-solving methods and techniques, they might end up analyzing the problem incorrectly, and choose the wrong solution. **Syntax**. Occur when a requirement author misunderstands the grammatical rules of natural language or the rules, symbols, or standards in a formal specification language like UML. **Information Management**. Result from a lack of knowledge about standard requirements engineering or documentation practices and procedures within the organization. **Wrong Assumptions**. Occur when the requirements author has a mistaken assumption about system features or stakeholder opinions. **Mistaken Belief that it is Impossible to Specify Non-Functional Requirements**. The requirements engineer(s) may believe that non-functional requirements cannot be captured and therefore omit this process from their elicitation and development plans. **Lack of Clear Distinction Between Client and Users**. If requirements engineering practitioners fail to distinguish between clients and end users, or do not realize that the clients are distinct from end users, they may fail to gather and analyze the end users' requirements. **Lack of Awareness of Requirement Sources**. Requirements gathering person is not aware of all stakeholders which he/she should contact in order to gather the complete set of user needs (including end users, customers, clients, and decision-makers). **Inappropriate Communication Based on Incomplete or Faulty Understanding of Rules**. Without proper understanding of developer roles, communication gaps may arise, either by failing to communicate at all or by ineffective communication. The management structure of project team resources is lacking. **Inadequate Requirements Process**. Occur when the requirement authors do not fully understand all of the requirements engineering steps necessary to ensure the software is complete and neglect to incorporate one or more essential steps into the plan.

Appendix H

Follow-Up Slides

This appendix contains the slides accompanying the optional follow-up survey (see Appendix I) sent to user study participants.



RIT

Human Error Assessment with a **Taxonomy of Human Errors in Software Engineering**

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 1

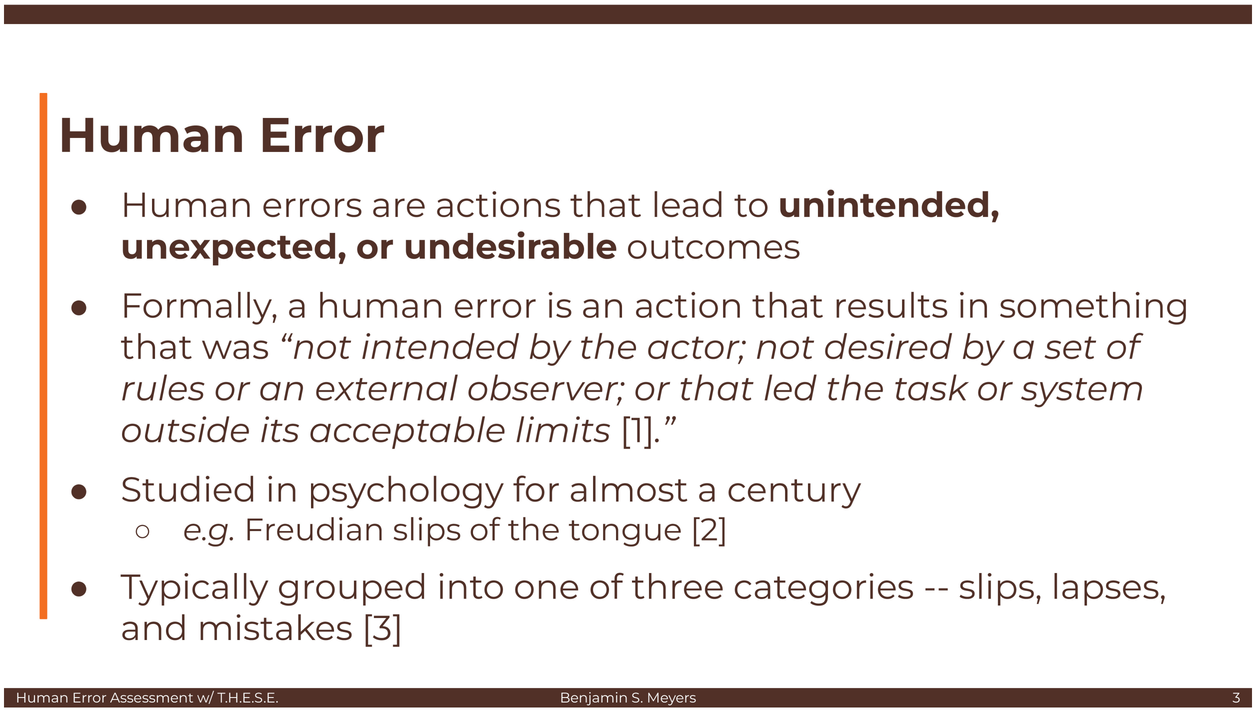
(a) Slide 1: Title slide

Human Error

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 2

(b) Slide 2: Section title for human error theory

Figure H.1: Optional Follow-Up Survey Slides

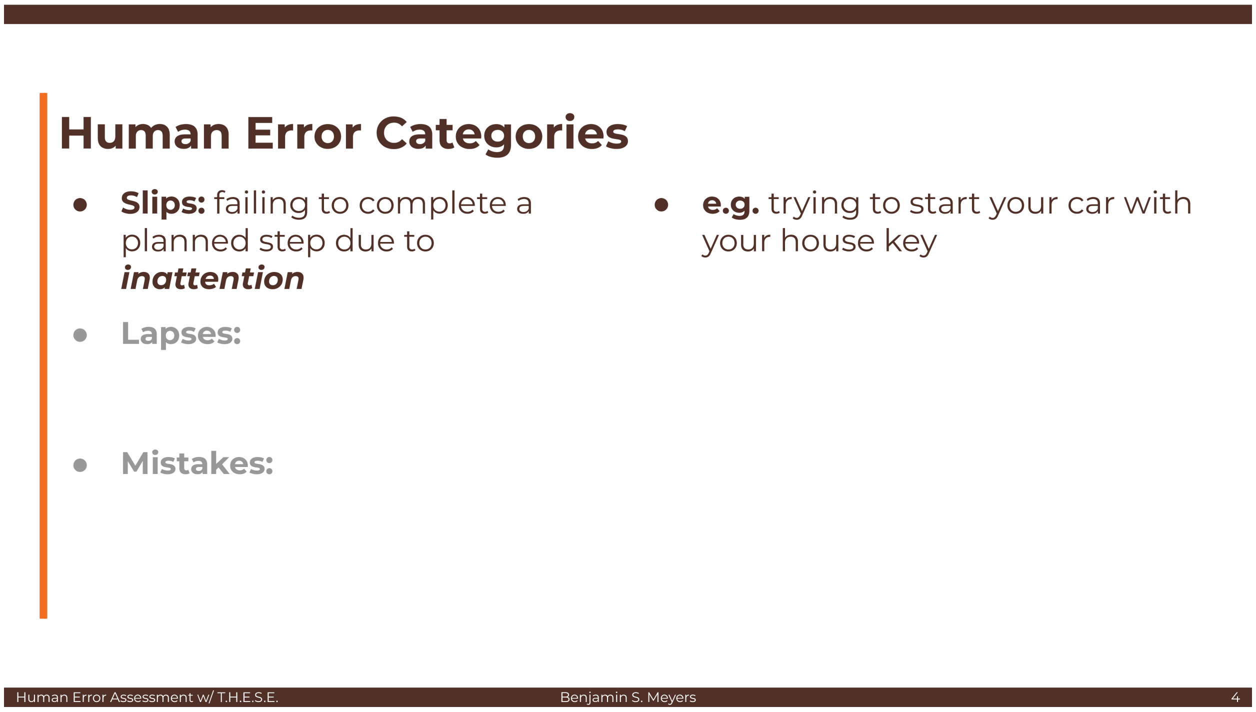


Human Error

- Human errors are actions that lead to **unintended, unexpected, or undesirable** outcomes
- Formally, a human error is an action that results in something that was *"not intended by the actor; not desired by a set of rules or an external observer; or that led the task or system outside its acceptable limits* [1]."
- Studied in psychology for almost a century
  - *e.g.* Freudian slips of the tongue [2]
- Typically grouped into one of three categories -- slips, lapses, and mistakes [3]

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 3

(c) Slide 3: Re-introduction to human error theory



Human Error Categories

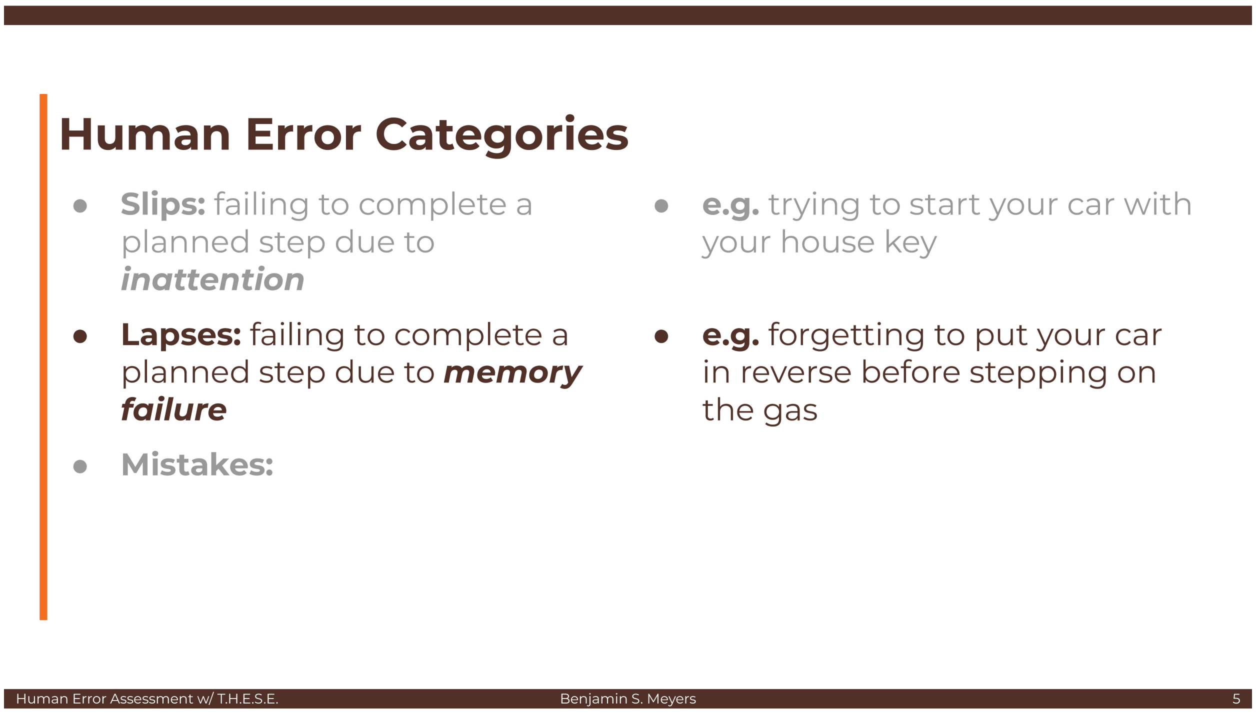
- **Slips:** failing to complete a planned step due to ***inattention***
- **Lapses:**
- **Mistakes:**

- **e.g.** trying to start your car with your house key

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 4

(d) Slide 4: Re-introduction to slips

Figure H.1: Optional Follow-Up Survey Slides (Continued)



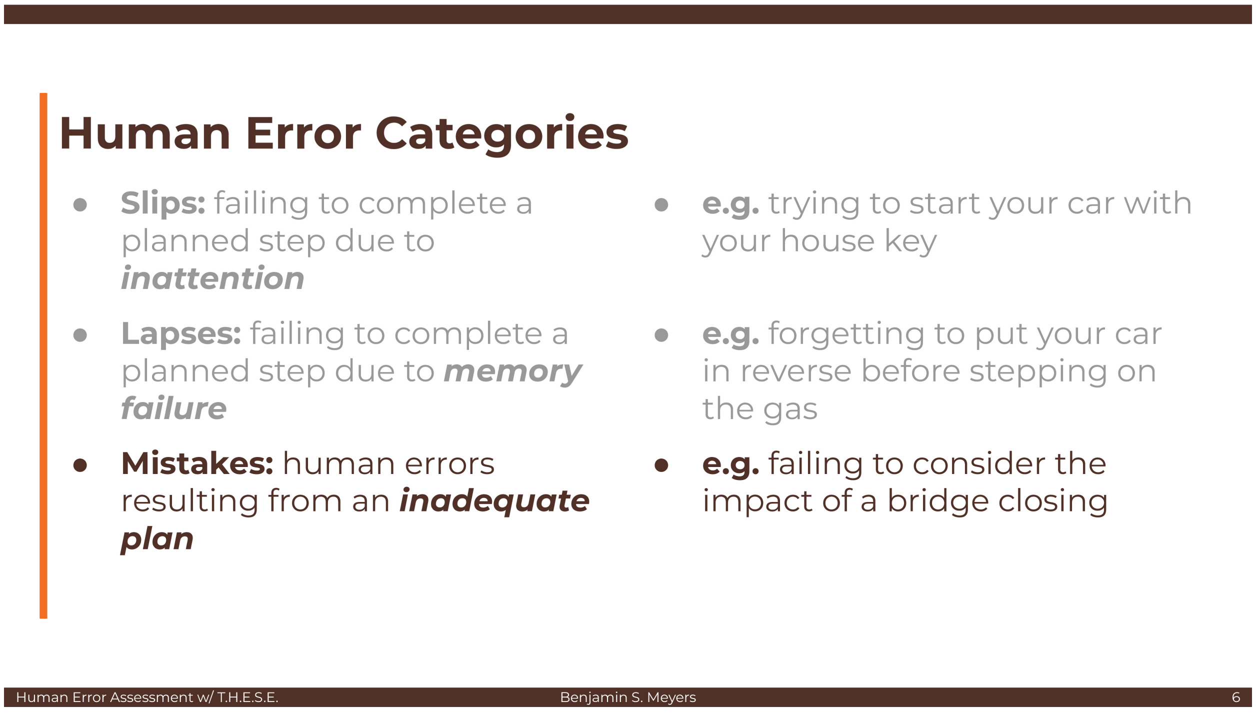
Human Error Categories

- **Slips:** failing to complete a planned step due to ***inattention***
- **Lapses:** failing to complete a planned step due to ***memory failure***
- **Mistakes:**

- **e.g.** trying to start your car with your house key
- **e.g.** forgetting to put your car in reverse before stepping on the gas

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 5

(e) Slide 5: Re-introduction to lapses



Human Error Categories

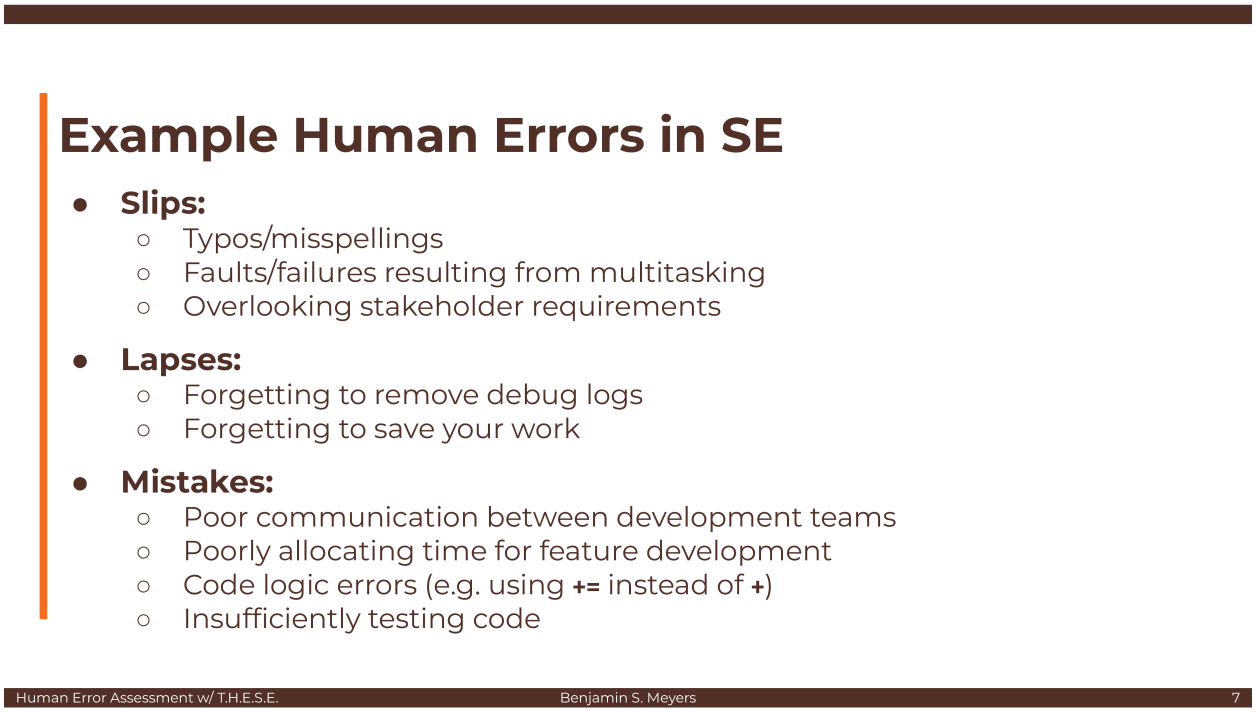
- **Slips:** failing to complete a planned step due to ***inattention***
- **Lapses:** failing to complete a planned step due to ***memory failure***
- **Mistakes:** human errors resulting from an ***inadequate plan***

- **e.g.** trying to start your car with your house key
- **e.g.** forgetting to put your car in reverse before stepping on the gas
- **e.g.** failing to consider the impact of a bridge closing

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 6

(f) Slide 6: Re-introduction to mistakes

Figure H.1: Optional Follow-Up Survey Slides (Continued)

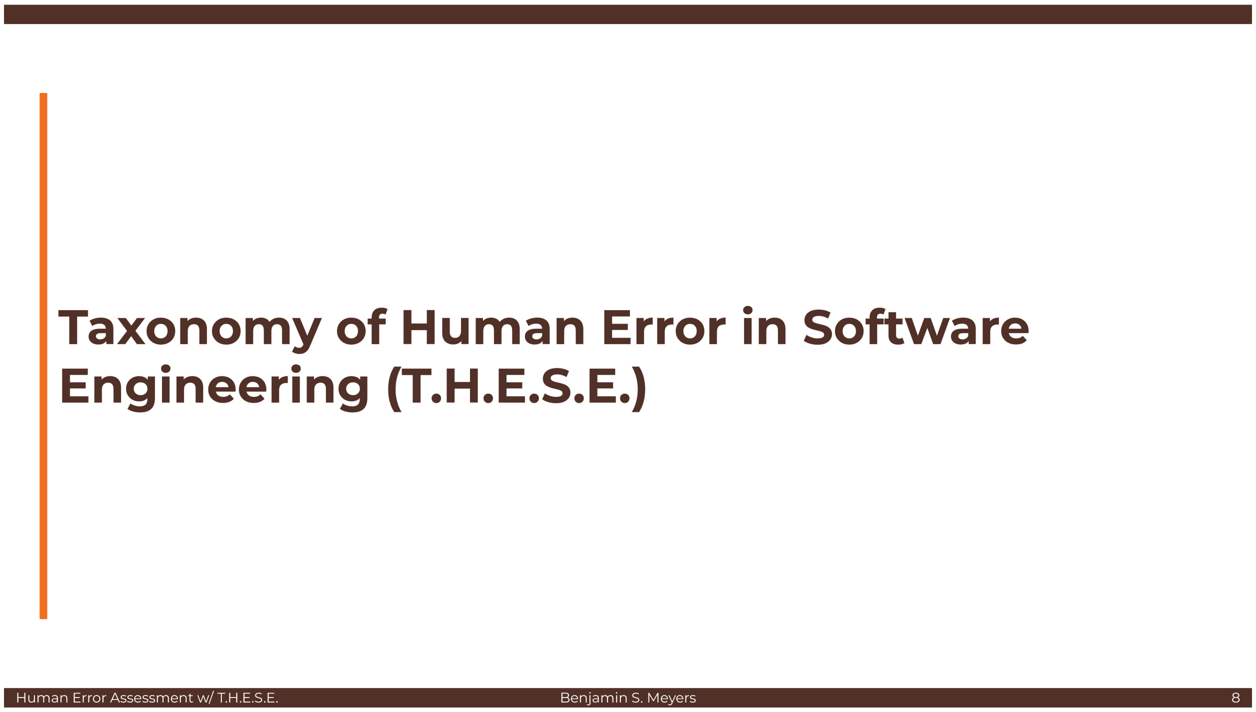


Example Human Errors in SE

- **Slips:**
  - Typos/misspellings
  - Faults/failures resulting from multitasking
  - Overlooking stakeholder requirements
- **Lapses:**
  - Forgetting to remove debug logs
  - Forgetting to save your work
- **Mistakes:**
  - Poor communication between development teams
  - Poorly allocating time for feature development
  - Code logic errors (e.g. using **+=** instead of **+**)
  - Insufficiently testing code

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 7

(g) Slide 7: Example human errors in software engineering

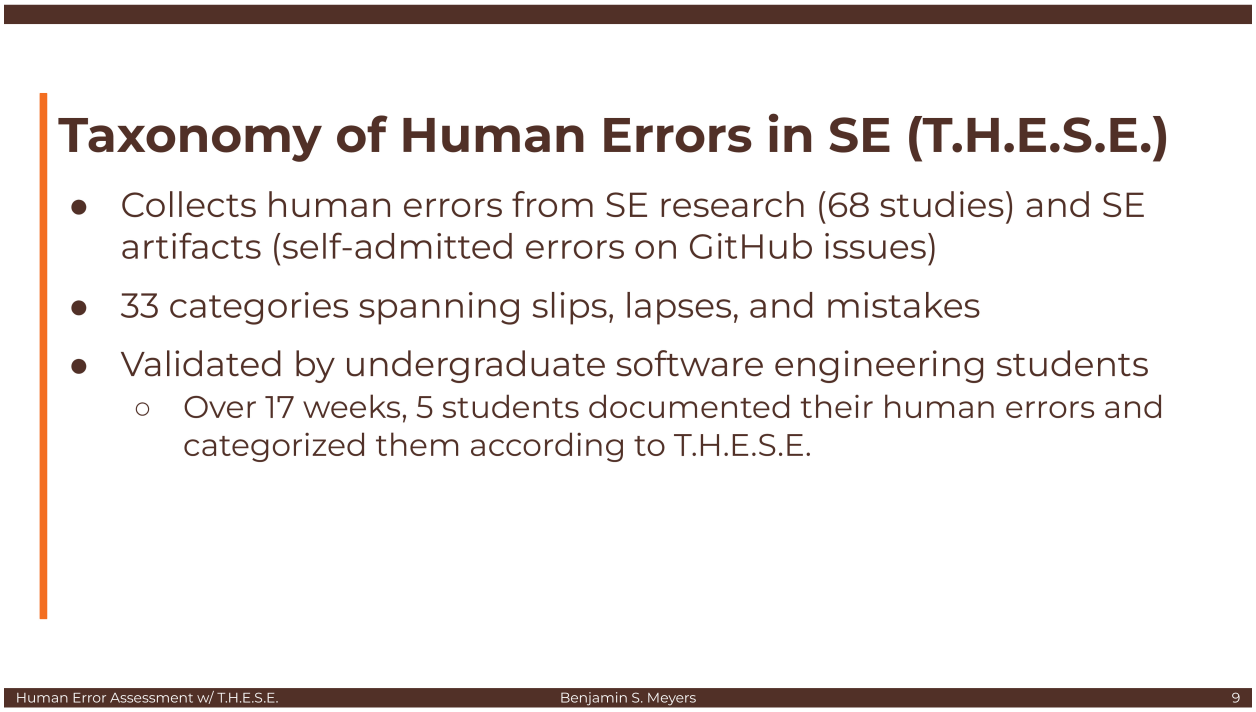


Taxonomy of Human Error in Software Engineering (T.H.E.S.E.)

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 8

(h) Slide 8: Section slide for T.H.E.S.E.

Figure H.1: Optional Follow-Up Survey Slides (Continued)

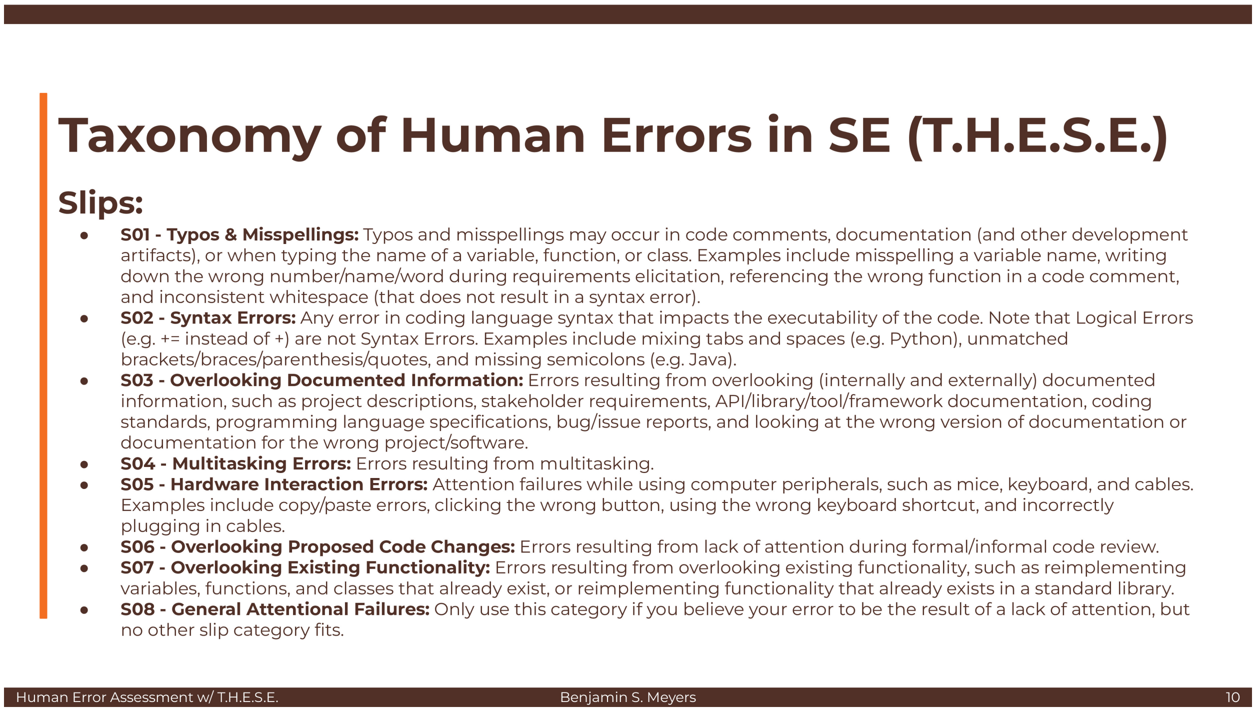


Taxonomy of Human Errors in SE (T.H.E.S.E.)

- Collects human errors from SE research (68 studies) and SE artifacts (self-admitted errors on GitHub issues)
- 33 categories spanning slips, lapses, and mistakes
- Validated by undergraduate software engineering students
  - Over 17 weeks, 5 students documented their human errors and categorized them according to T.H.E.S.E.

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 9

(i) Slide 9: Re-introduction to T.H.E.S.E.



Taxonomy of Human Errors in SE (T.H.E.S.E.)

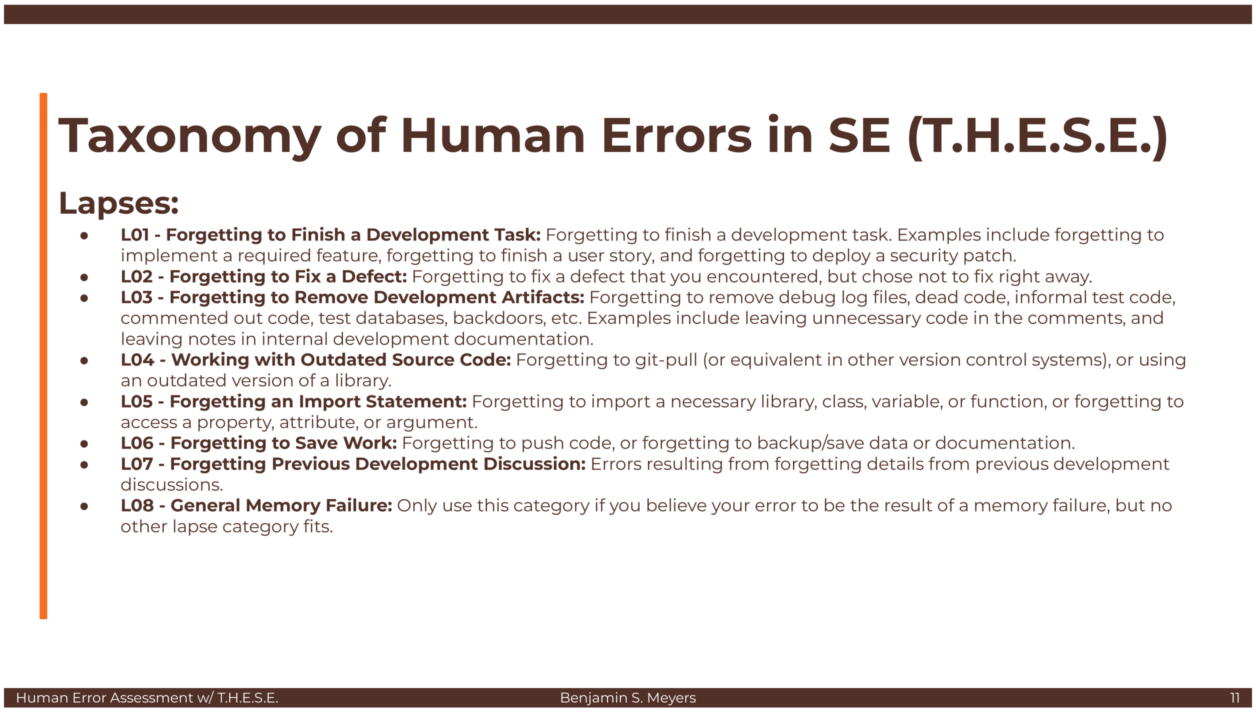
Slips:

- **S01 - Typos & Misspellings:** Typos and misspellings may occur in code comments, documentation (and other development artifacts), or when typing the name of a variable, function, or class. Examples include misspelling a variable name, writing down the wrong number/name/word during requirements elicitation, referencing the wrong function in a code comment, and inconsistent whitespace (that does not result in a syntax error).
- **S02 - Syntax Errors:** Any error in coding language syntax that impacts the executability of the code. Note that Logical Errors (e.g. += instead of +) are not Syntax Errors. Examples include mixing tabs and spaces (e.g. Python), unmatched brackets/braces/parenthesis/quotes, and missing semicolons (e.g. Java).
- **S03 - Overlooking Documented Information:** Errors resulting from overlooking (internally and externally) documented information, such as project descriptions, stakeholder requirements, API/library/tool/framework documentation, coding standards, programming language specifications, bug/issue reports, and looking at the wrong version of documentation or documentation for the wrong project/software.
- **S04 - Multitasking Errors:** Errors resulting from multitasking.
- **S05 - Hardware Interaction Errors:** Attention failures while using computer peripherals, such as mice, keyboard, and cables. Examples include copy/paste errors, clicking the wrong button, using the wrong keyboard shortcut, and incorrectly plugging in cables.
- **S06 - Overlooking Proposed Code Changes:** Errors resulting from lack of attention during formal/informal code review.
- **S07 - Overlooking Existing Functionality:** Errors resulting from overlooking existing functionality, such as reimplementing variables, functions, and classes that already exist, or reimplementing functionality that already exists in a standard library.
- **S08 - General Attentional Failures:** Only use this category if you believe your error to be the result of a lack of attention, but no other slip category fits.

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 10

(j) Slide 10: Review of slip categories

Figure H.1: Optional Follow-Up Survey Slides (Continued)



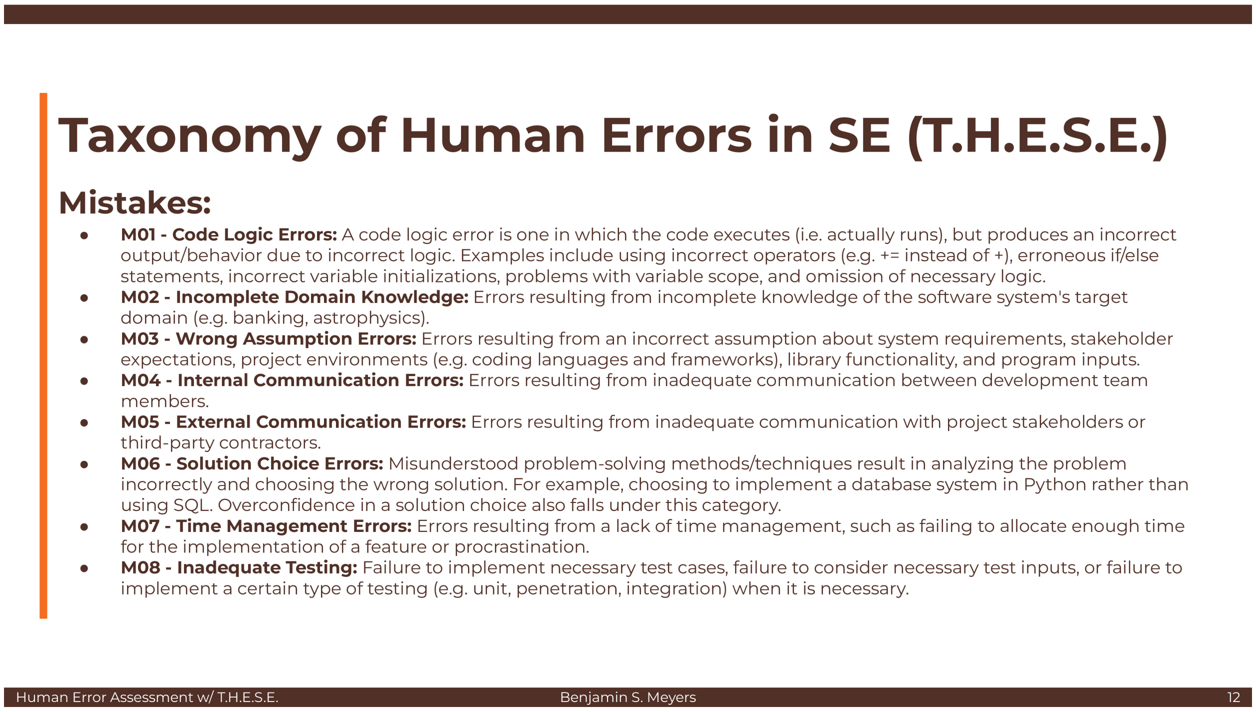
Taxonomy of Human Errors in SE (T.H.E.S.E.)

Lapses:

- **L01 - Forgetting to Finish a Development Task:** Forgetting to finish a development task. Examples include forgetting to implement a required feature, forgetting to finish a user story, and forgetting to deploy a security patch.
- **L02 - Forgetting to Fix a Defect:** Forgetting to fix a defect that you encountered, but chose not to fix right away.
- **L03 - Forgetting to Remove Development Artifacts:** Forgetting to remove debug log files, dead code, informal test code, commented out code, test databases, backdoors, etc. Examples include leaving unnecessary code in the comments, and leaving notes in internal development documentation.
- **L04 - Working with Outdated Source Code:** Forgetting to git-pull (or equivalent in other version control systems), or using an outdated version of a library.
- **L05 - Forgetting an Import Statement:** Forgetting to import a necessary library, class, variable, or function, or forgetting to access a property, attribute, or argument.
- **L06 - Forgetting to Save Work:** Forgetting to push code, or forgetting to backup/save data or documentation.
- **L07 - Forgetting Previous Development Discussion:** Errors resulting from forgetting details from previous development discussions.
- **L08 - General Memory Failure:** Only use this category if you believe your error to be the result of a memory failure, but no other lapse category fits.

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 11

(k) Slide 11: Review of lapse categories



Taxonomy of Human Errors in SE (T.H.E.S.E.)

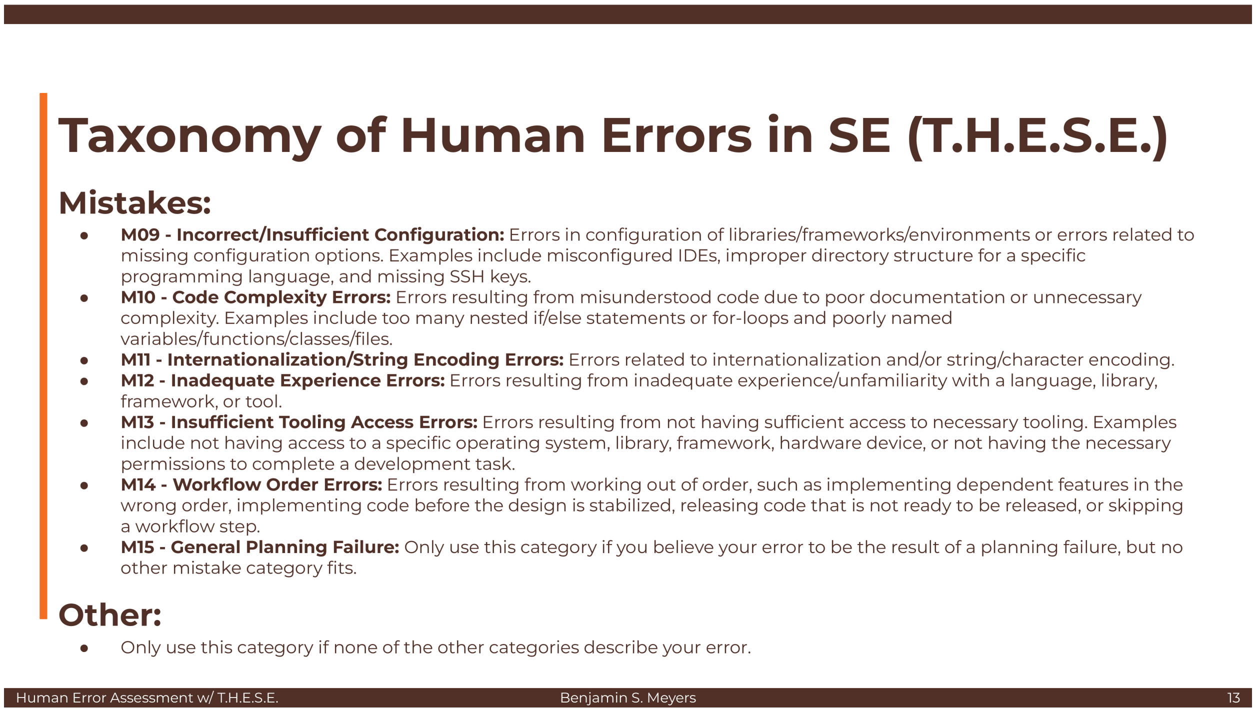
Mistakes:

- **M01 - Code Logic Errors:** A code logic error is one in which the code executes (i.e. actually runs), but produces an incorrect output/behavior due to incorrect logic. Examples include using incorrect operators (e.g. += instead of +), erroneous if/else statements, incorrect variable initializations, problems with variable scope, and omission of necessary logic.
- **M02 - Incomplete Domain Knowledge:** Errors resulting from incomplete knowledge of the software system's target domain (e.g. banking, astrophysics).
- **M03 - Wrong Assumption Errors:** Errors resulting from an incorrect assumption about system requirements, stakeholder expectations, project environments (e.g. coding languages and frameworks), library functionality, and program inputs.
- **M04 - Internal Communication Errors:** Errors resulting from inadequate communication between development team members.
- **M05 - External Communication Errors:** Errors resulting from inadequate communication with project stakeholders or third-party contractors.
- **M06 - Solution Choice Errors:** Misunderstood problem-solving methods/techniques result in analyzing the problem incorrectly and choosing the wrong solution. For example, choosing to implement a database system in Python rather than using SQL. Overconfidence in a solution choice also falls under this category.
- **M07 - Time Management Errors:** Errors resulting from a lack of time management, such as failing to allocate enough time for the implementation of a feature or procrastination.
- **M08 - Inadequate Testing:** Failure to implement necessary test cases, failure to consider necessary test inputs, or failure to implement a certain type of testing (e.g. unit, penetration, integration) when it is necessary.

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 12

(l) Slide 12: Review of mistake categories

Figure H.1: Optional Follow-Up Survey Slides (Continued)



Taxonomy of Human Errors in SE (T.H.E.S.E.)

Mistakes:

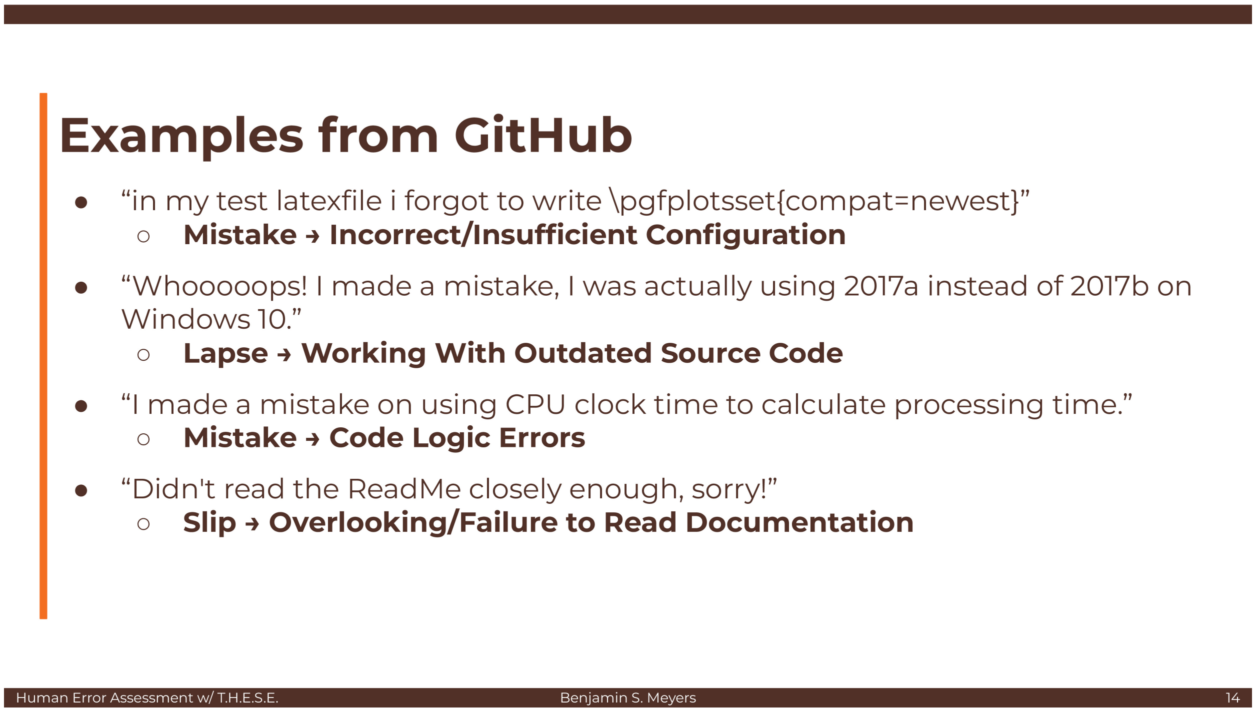
- **M09 - Incorrect/Insufficient Configuration:** Errors in configuration of libraries/frameworks/environments or errors related to missing configuration options. Examples include misconfigured IDEs, improper directory structure for a specific programming language, and missing SSH keys.
- **M10 - Code Complexity Errors:** Errors resulting from misunderstood code due to poor documentation or unnecessary complexity. Examples include too many nested if/else statements or for-loops and poorly named variables/functions/classes/files.
- **M11 - Internationalization/String Encoding Errors:** Errors related to internationalization and/or string/character encoding.
- **M12 - Inadequate Experience Errors:** Errors resulting from inadequate experience/unfamiliarity with a language, library, framework, or tool.
- **M13 - Insufficient Tooling Access Errors:** Errors resulting from not having sufficient access to necessary tooling. Examples include not having access to a specific operating system, library, framework, hardware device, or not having the necessary permissions to complete a development task.
- **M14 - Workflow Order Errors:** Errors resulting from working out of order, such as implementing dependent features in the wrong order, implementing code before the design is stabilized, releasing code that is not ready to be released, or skipping a workflow step.
- **M15 - General Planning Failure:** Only use this category if you believe your error to be the result of a planning failure, but no other mistake category fits.

Other:

- Only use this category if none of the other categories describe your error.

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 13

(m) Slide 13: Review of mistake categories (continued)



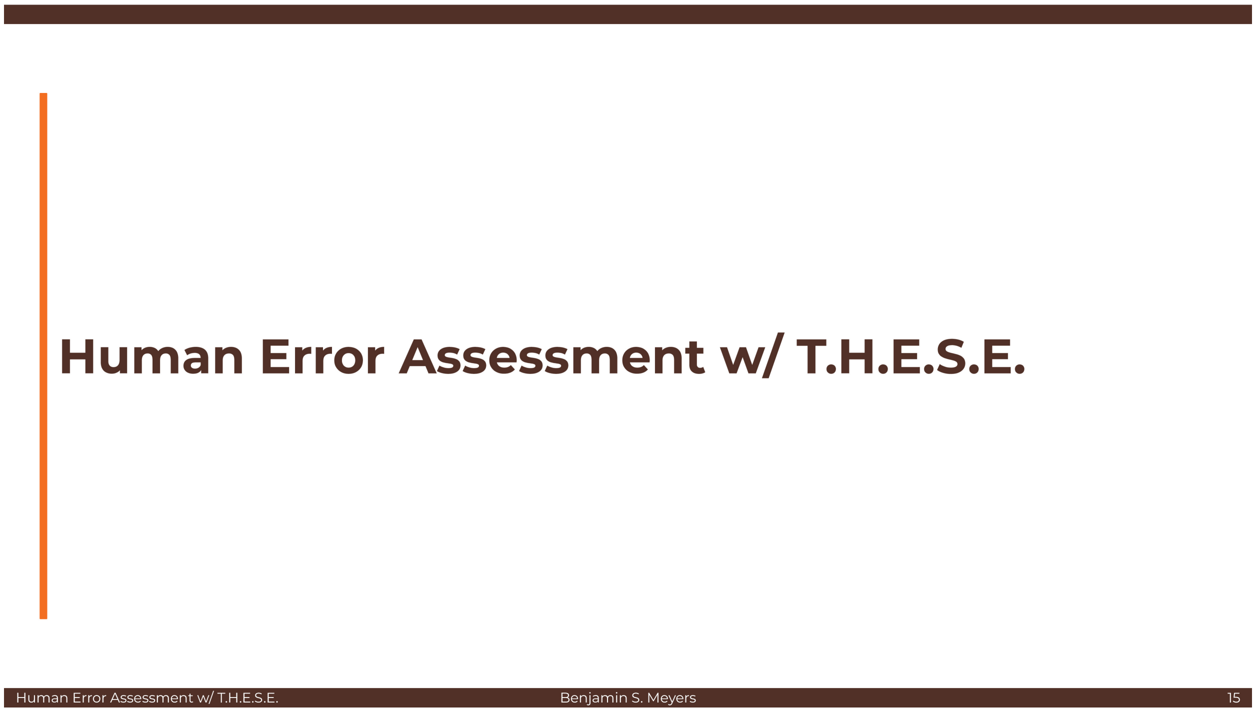
Examples from GitHub

- “in my test latexfile i forgot to write \pgfplotsset{compat=newest}”
  - **Mistake → Incorrect/Insufficient Configuration**
- “Whooooops! I made a mistake, I was actually using 2017a instead of 2017b on Windows 10.”
  - **Lapse → Working With Outdated Source Code**
- “I made a mistake on using CPU clock time to calculate processing time.”
  - **Mistake → Code Logic Errors**
- “Didn't read the ReadMe closely enough, sorry!”
  - **Slip → Overlooking/Failure to Read Documentation**

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 14

(n) Slide 14: Example human errors from GitHub with categorizations

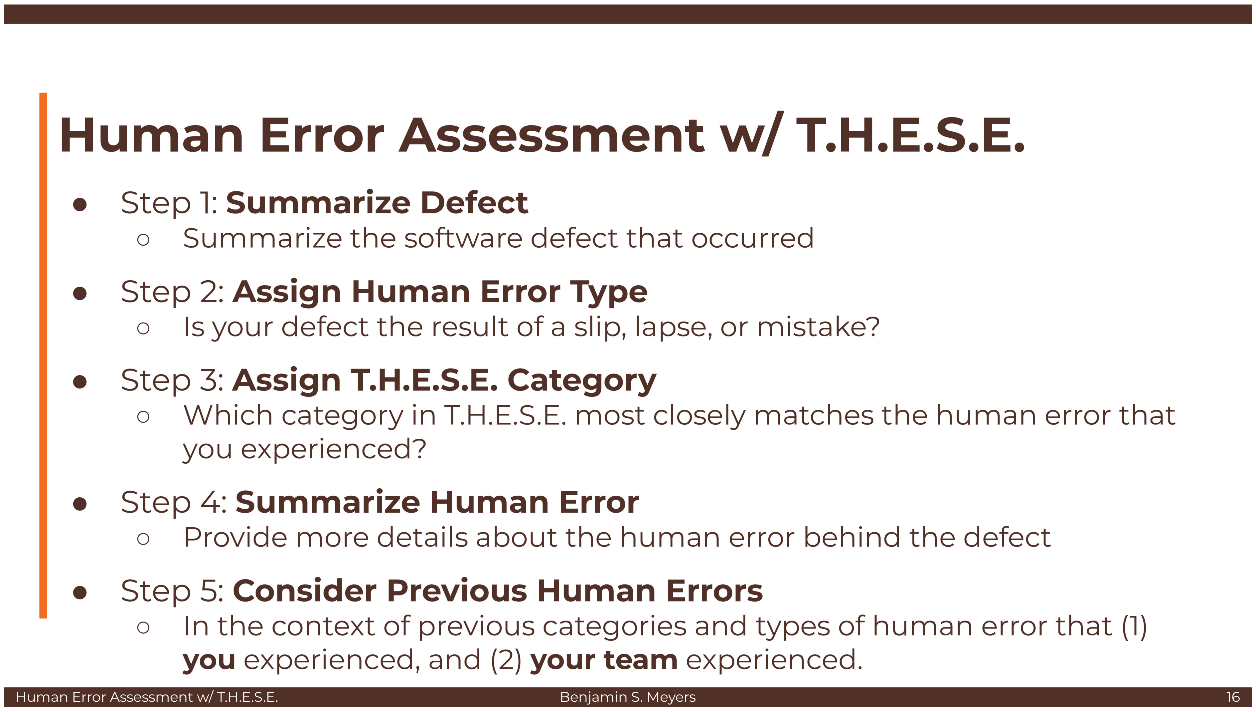
Figure H.1: Optional Follow-Up Survey Slides (Continued)



Human Error Assessment w/ T.H.E.S.E.

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 15

(o) Slide 15: Section title for human error assessment



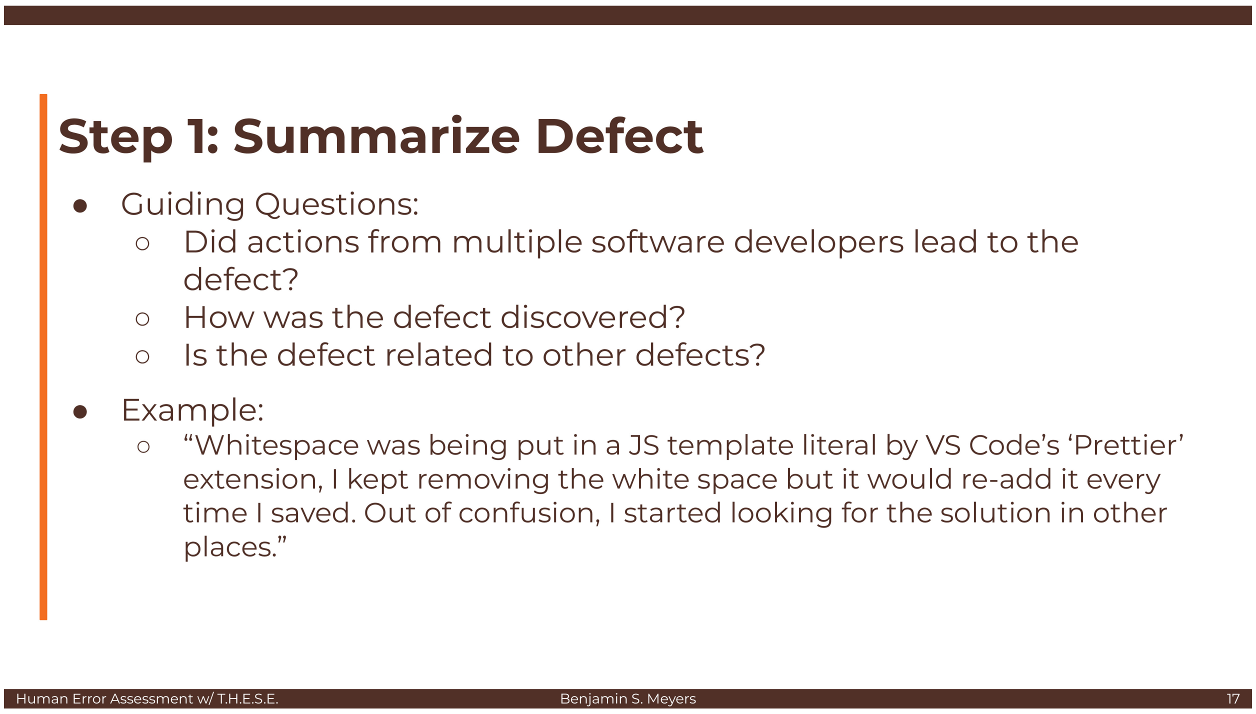
Human Error Assessment w/ T.H.E.S.E.

- Step 1: **Summarize Defect**
  - Summarize the software defect that occurred
- Step 2: **Assign Human Error Type**
  - Is your defect the result of a slip, lapse, or mistake?
- Step 3: **Assign T.H.E.S.E. Category**
  - Which category in T.H.E.S.E. most closely matches the human error that you experienced?
- Step 4: **Summarize Human Error**
  - Provide more details about the human error behind the defect
- Step 5: **Consider Previous Human Errors**
  - In the context of previous categories and types of human error that (1) **you** experienced, and (2) **your team** experienced.

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 16

(p) Slide 16: Summary of human error informed micro post-mortem process

Figure H.1: Optional Follow-Up Survey Slides (Continued)

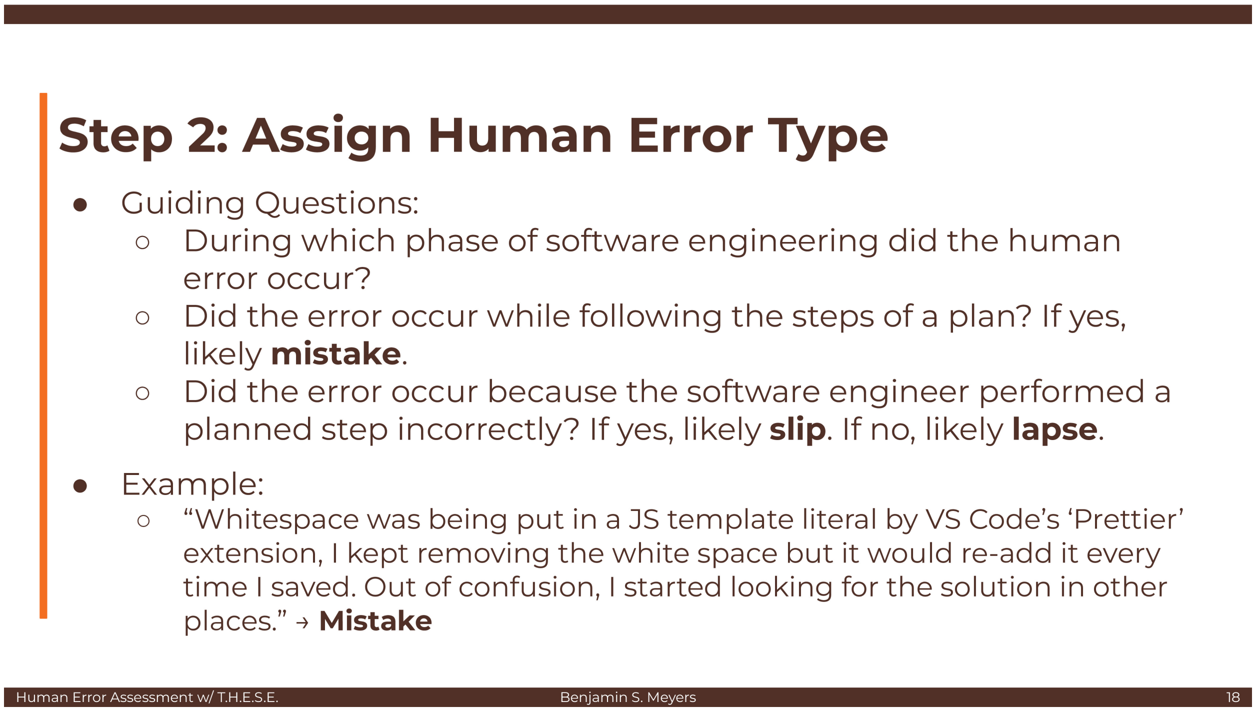


Step 1: Summarize Defect

- Guiding Questions:
  - Did actions from multiple software developers lead to the defect?
  - How was the defect discovered?
  - Is the defect related to other defects?
- Example:
  - "Whitespace was being put in a JS template literal by VS Code's 'Prettier' extension, I kept removing the white space but it would re-add it every time I saved. Out of confusion, I started looking for the solution in other places."

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 17

(q) Slide 17: Step 1 details with example



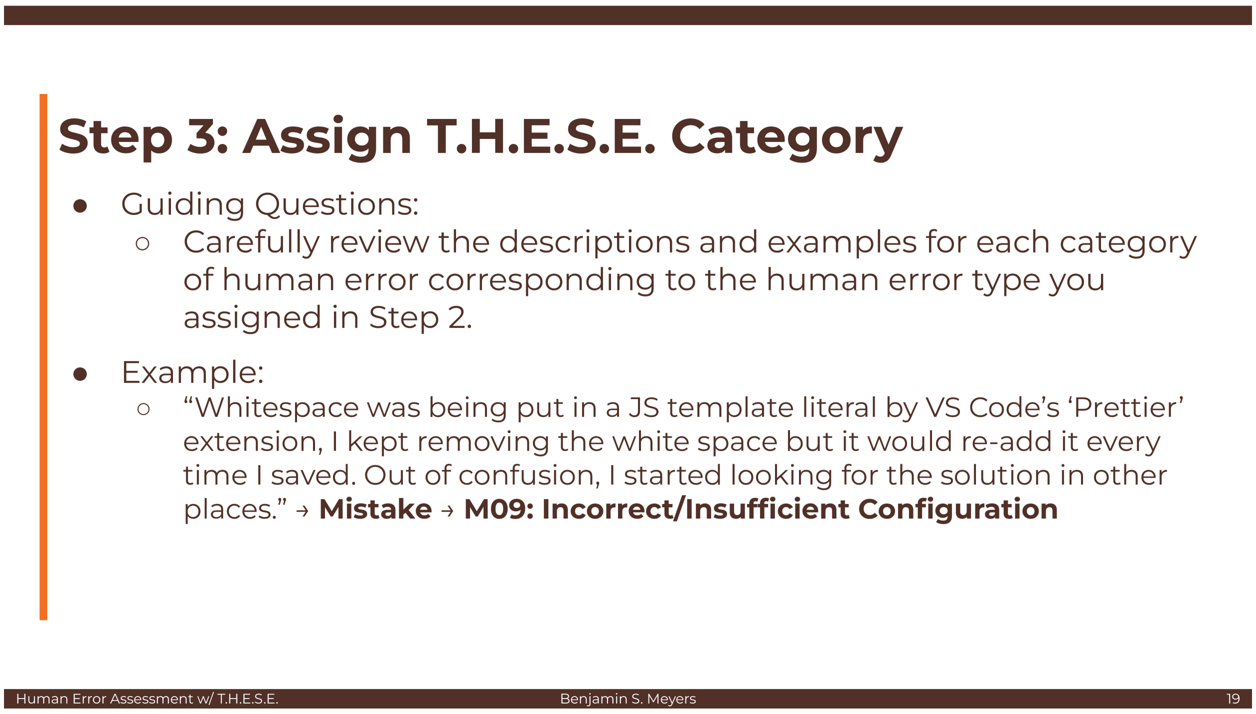
Step 2: Assign Human Error Type

- Guiding Questions:
  - During which phase of software engineering did the human error occur?
  - Did the error occur while following the steps of a plan? If yes, likely **mistake**.
  - Did the error occur because the software engineer performed a planned step incorrectly? If yes, likely **slip**. If no, likely **lapse**.
- Example:
  - "Whitespace was being put in a JS template literal by VS Code's 'Prettier' extension, I kept removing the white space but it would re-add it every time I saved. Out of confusion, I started looking for the solution in other places." → **Mistake**

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 18

(r) Slide 18: Step 2 details with example

Figure H.1: Optional Follow-Up Survey Slides (Continued)

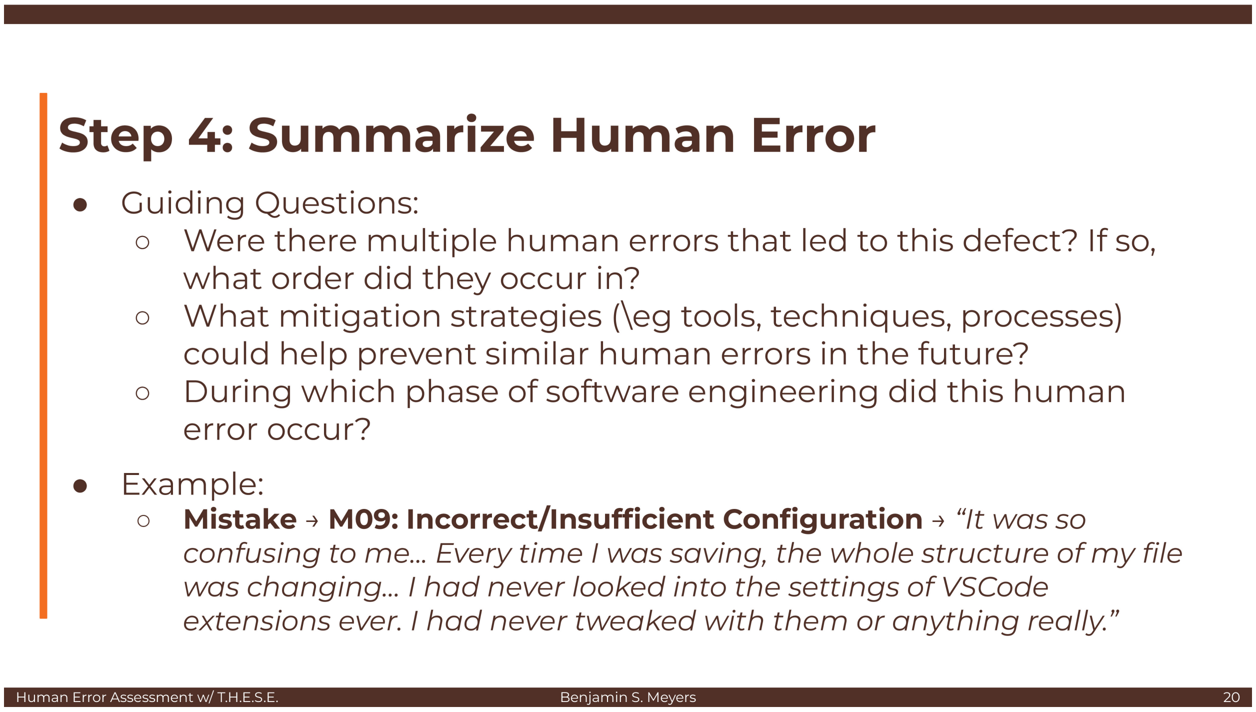


Step 3: Assign T.H.E.S.E. Category

- Guiding Questions:
  - Carefully review the descriptions and examples for each category of human error corresponding to the human error type you assigned in Step 2.
- Example:
  - "Whitespace was being put in a JS template literal by VS Code's 'Prettier' extension, I kept removing the white space but it would re-add it every time I saved. Out of confusion, I started looking for the solution in other places." → **Mistake** → **M09: Incorrect/Insufficient Configuration**

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 19

(s) Slide 19: Step 3 details with example



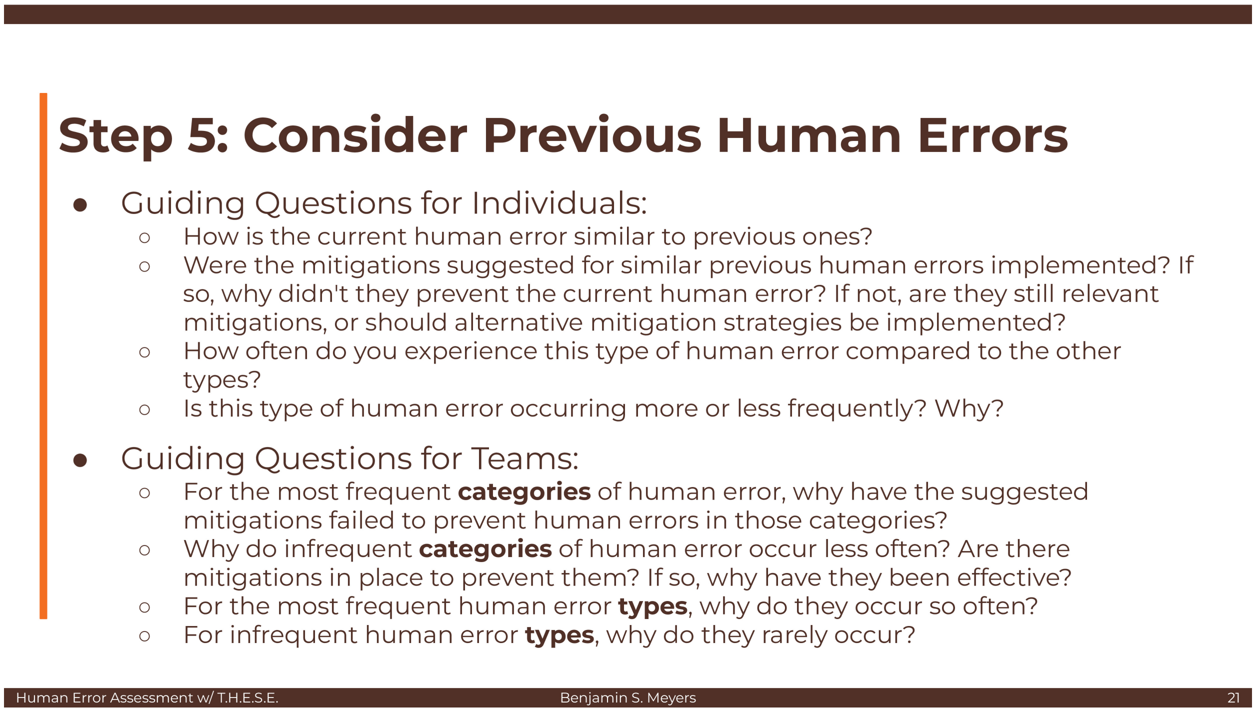
Step 4: Summarize Human Error

- Guiding Questions:
  - Were there multiple human errors that led to this defect? If so, what order did they occur in?
  - What mitigation strategies (\eg tools, techniques, processes) could help prevent similar human errors in the future?
  - During which phase of software engineering did this human error occur?
- Example:
  - **Mistake** → **M09: Incorrect/Insufficient Configuration** → *"It was so confusing to me... Every time I was saving, the whole structure of my file was changing... I had never looked into the settings of VSCode extensions ever. I had never tweaked with them or anything really."*

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 20

(t) Slide 20: Step 4 details with example

Figure H.1: Optional Follow-Up Survey Slides (Continued)

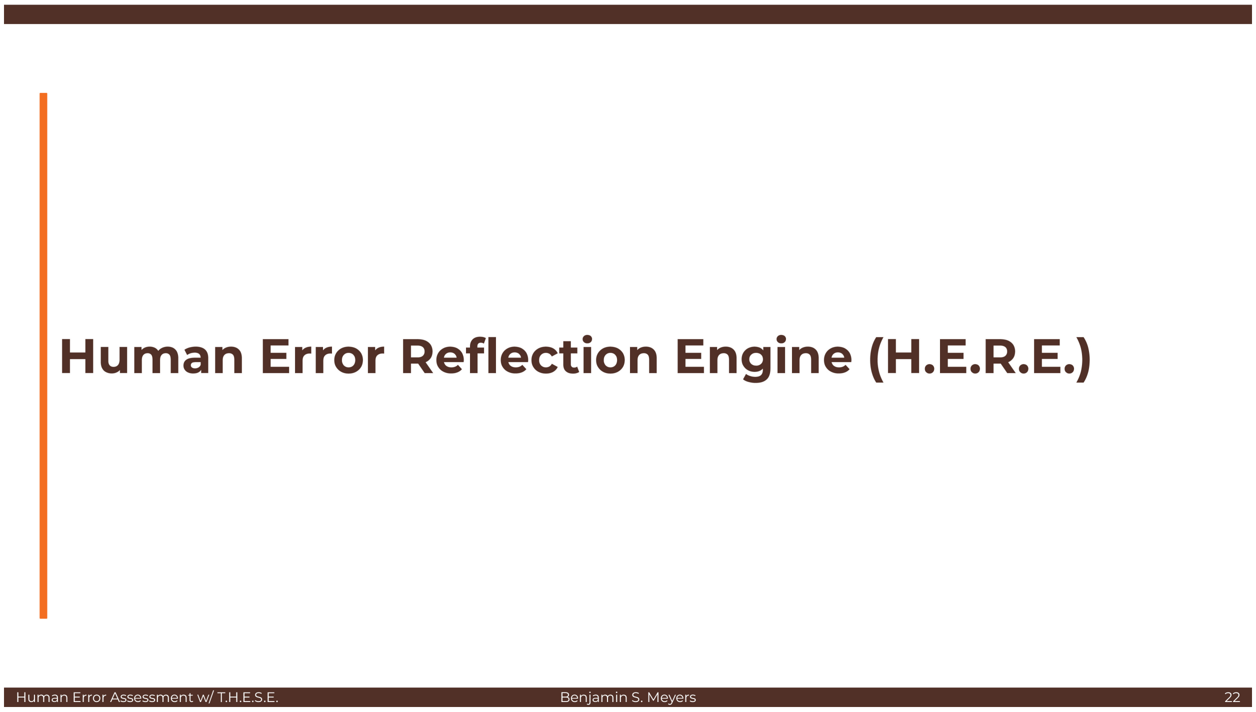


Step 5: Consider Previous Human Errors

- Guiding Questions for Individuals:
  - How is the current human error similar to previous ones?
  - Were the mitigations suggested for similar previous human errors implemented? If so, why didn't they prevent the current human error? If not, are they still relevant mitigations, or should alternative mitigation strategies be implemented?
  - How often do you experience this type of human error compared to the other types?
  - Is this type of human error occurring more or less frequently? Why?
- Guiding Questions for Teams:
  - For the most frequent **categories** of human error, why have the suggested mitigations failed to prevent human errors in those categories?
  - Why do infrequent **categories** of human error occur less often? Are there mitigations in place to prevent them? If so, why have they been effective?
  - For the most frequent human error **types**, why do they occur so often?
  - For infrequent human error **types**, why do they rarely occur?

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 21

(u) Slide 21: Step 5 details

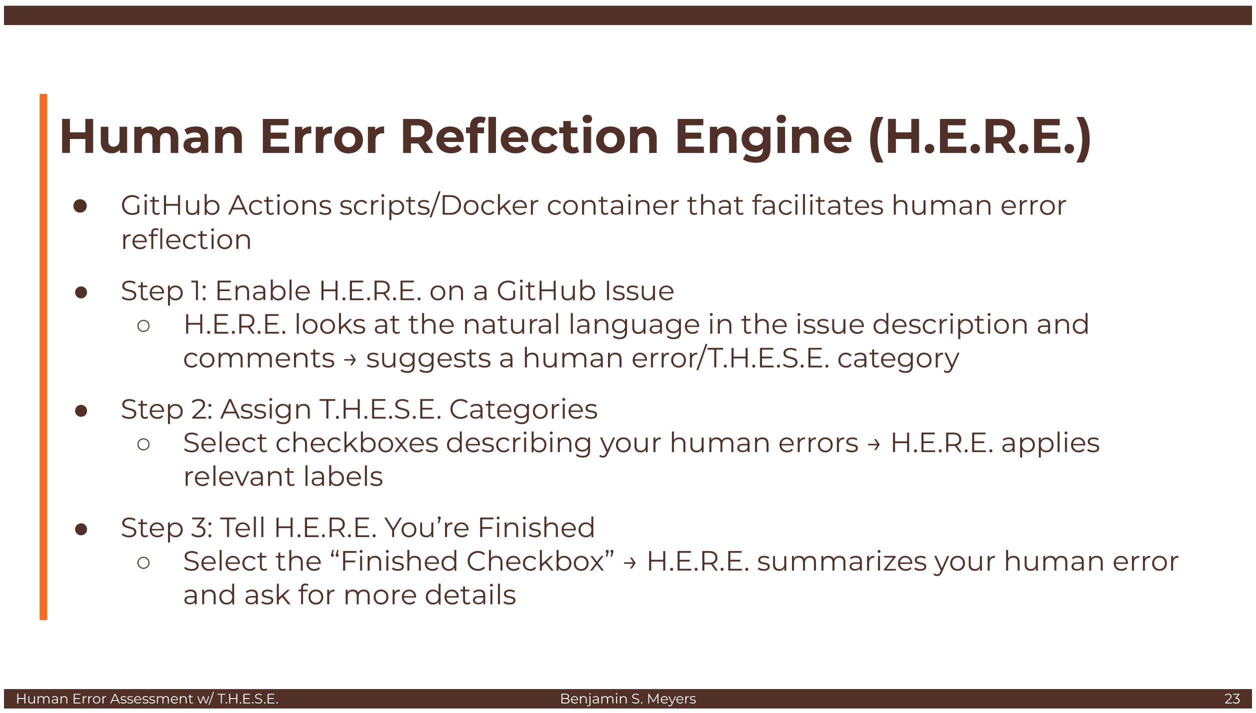


Human Error Reflection Engine (H.E.R.E.)

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 22

(v) Slide 22: Section title for H.E.R.E.

Figure H.1: Optional Follow-Up Survey Slides (Continued)

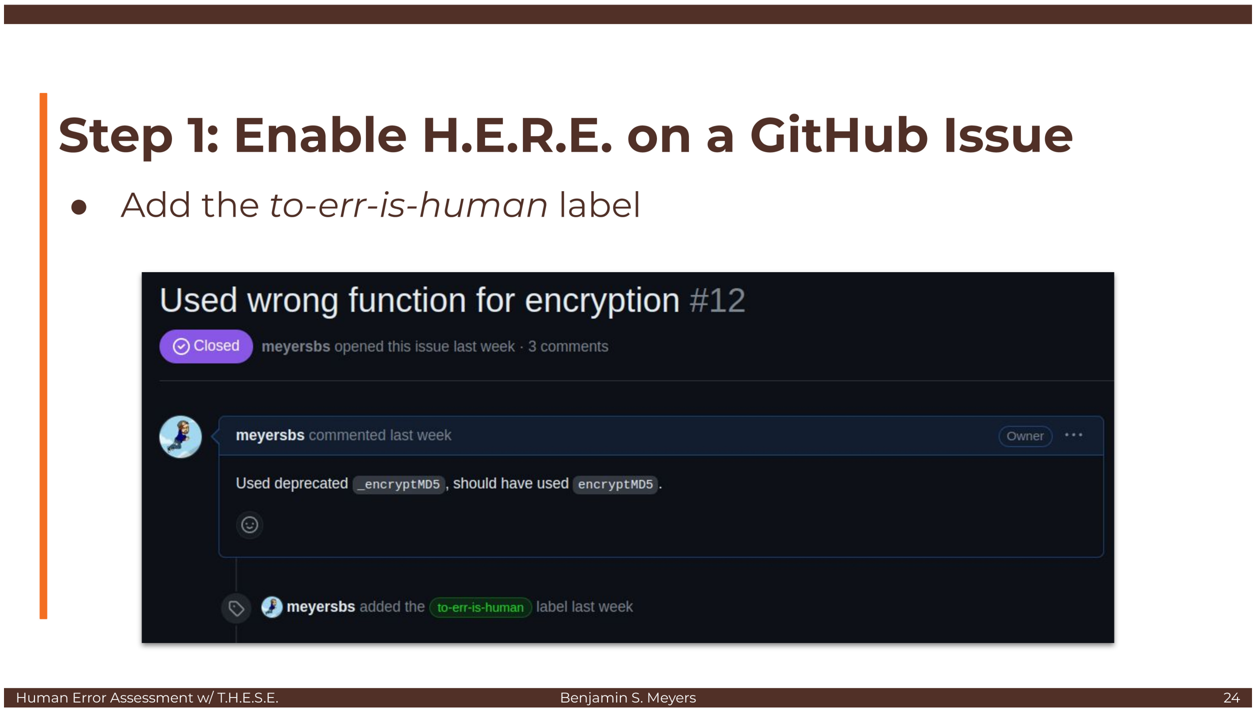


Human Error Reflection Engine (H.E.R.E.)

- GitHub Actions scripts/Docker container that facilitates human error reflection
- Step 1: Enable H.E.R.E. on a GitHub Issue
  - H.E.R.E. looks at the natural language in the issue description and comments → suggests a human error/T.H.E.S.E. category
- Step 2: Assign T.H.E.S.E. Categories
  - Select checkboxes describing your human errors → H.E.R.E. applies relevant labels
- Step 3: Tell H.E.R.E. You're Finished
  - Select the "Finished Checkbox" → H.E.R.E. summarizes your human error and ask for more details

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 23

(w) Slide 23: Introduction to H.E.R.E.



Step 1: Enable H.E.R.E. on a GitHub Issue

- Add the *to-err-is-human* label

Used wrong function for encryption #12

Closed meyersbs opened this issue last week · 3 comments

meyersbs commented last week Owner

Used deprecated `_encryptMD5`, should have used `encryptMD5`.

meyersbs added the to-err-is-human label last week

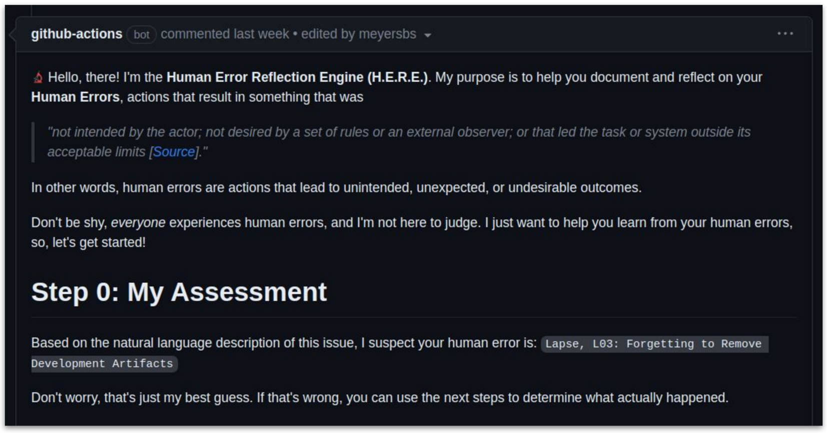
Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 24

(x) Slide 24: H.E.R.E. step 1 with screenshot

Figure H.1: Optional Follow-Up Survey Slides (Continued)

Step 1: Enable H.E.R.E. on a GitHub Issue

- H.E.R.E. responds with suggested human error



github-actions bot commented last week • edited by meyersbs

Hello, there! I'm the **Human Error Reflection Engine (H.E.R.E.)**. My purpose is to help you document and reflect on your **Human Errors**, actions that result in something that was

> *"not intended by the actor; not desired by a set of rules or an external observer; or that led the task or system outside its acceptable limits [Source]."*

In other words, human errors are actions that lead to unintended, unexpected, or undesirable outcomes.

Don't be shy, *everyone* experiences human errors, and I'm not here to judge. I just want to help you learn from your human errors, so, let's get started!

Step 0: My Assessment

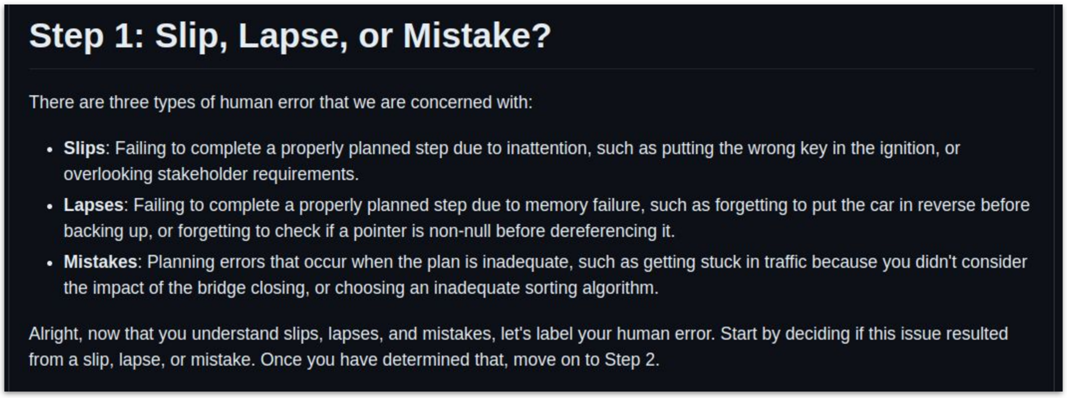
Based on the natural language description of this issue, I suspect your human error is: `Lapse, L03: Forgetting to Remove Development Artifacts`

Don't worry, that's just my best guess. If that's wrong, you can use the next steps to determine what actually happened.

(y) Slide 25: H.E.R.E. step 1 response with screenshot

Step 2: Assign T.H.E.S.E. Categories

- Assign slip, lapse, or mistake



Step 1: Slip, Lapse, or Mistake?

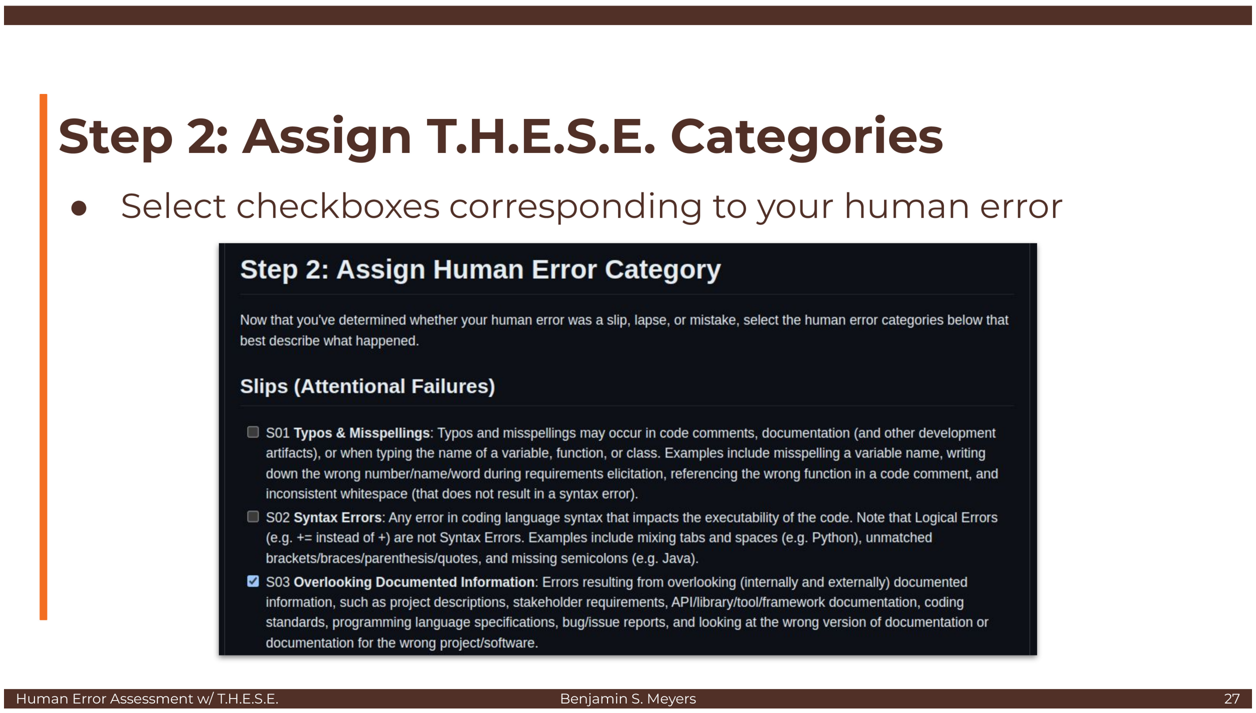
There are three types of human error that we are concerned with:

- **Slips**: Failing to complete a properly planned step due to inattention, such as putting the wrong key in the ignition, or overlooking stakeholder requirements.
- **Lapses**: Failing to complete a properly planned step due to memory failure, such as forgetting to put the car in reverse before backing up, or forgetting to check if a pointer is non-null before dereferencing it.
- **Mistakes**: Planning errors that occur when the plan is inadequate, such as getting stuck in traffic because you didn't consider the impact of the bridge closing, or choosing an inadequate sorting algorithm.

Alright, now that you understand slips, lapses, and mistakes, let's label your human error. Start by deciding if this issue resulted from a slip, lapse, or mistake. Once you have determined that, move on to Step 2.

(z) Slide 26: H.E.R.E. step 2 with screenshot

Figure H.1: Optional Follow-Up Survey Slides (Continued)



Step 2: Assign T.H.E.S.E. Categories

- Select checkboxes corresponding to your human error

Step 2: Assign Human Error Category

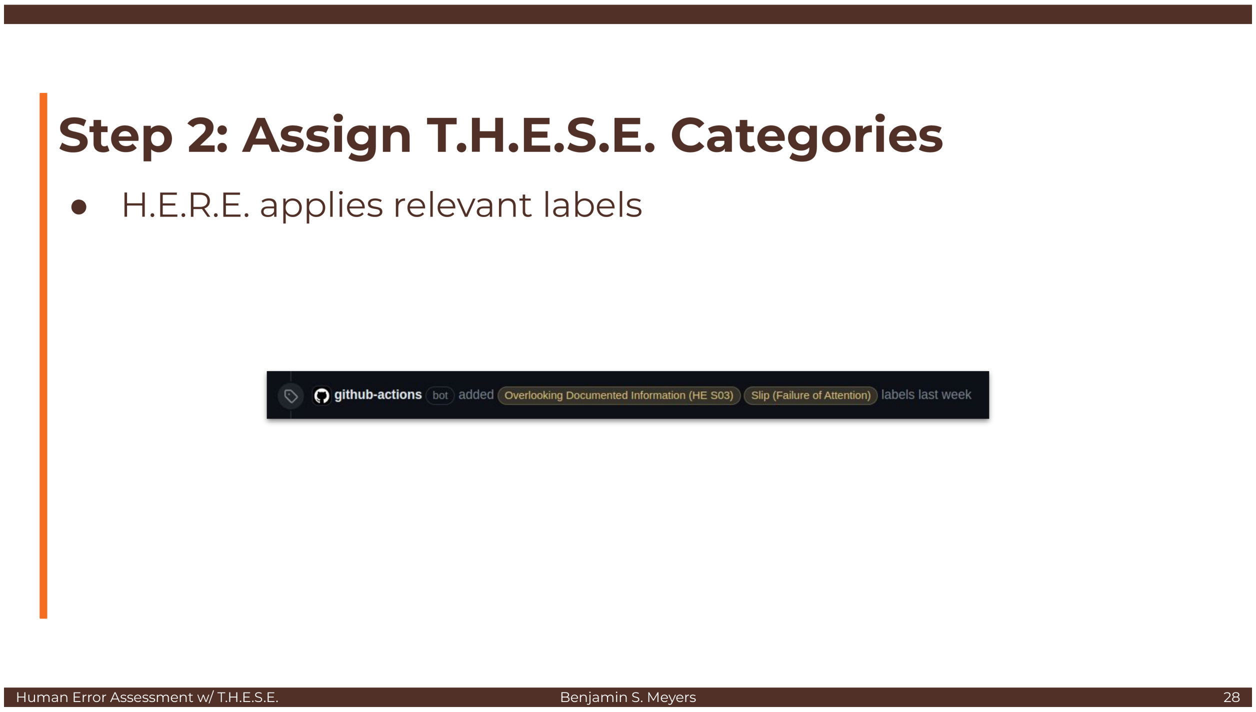
Now that you've determined whether your human error was a slip, lapse, or mistake, select the human error categories below that best describe what happened.

Slips (Attentional Failures)

- [ ] S01 **Typos & Misspellings**: Typos and misspellings may occur in code comments, documentation (and other development artifacts), or when typing the name of a variable, function, or class. Examples include misspelling a variable name, writing down the wrong number/name/word during requirements elicitation, referencing the wrong function in a code comment, and inconsistent whitespace (that does not result in a syntax error).
- [ ] S02 **Syntax Errors**: Any error in coding language syntax that impacts the executability of the code. Note that Logical Errors (e.g. += instead of +) are not Syntax Errors. Examples include mixing tabs and spaces (e.g. Python), unmatched brackets/braces/parenthesis/quotes, and missing semicolons (e.g. Java).
- [x] S03 **Overlooking Documented Information**: Errors resulting from overlooking (internally and externally) documented information, such as project descriptions, stakeholder requirements, API/library/tool/framework documentation, coding standards, programming language specifications, bug/issue reports, and looking at the wrong version of documentation or documentation for the wrong project/software.

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 27

(aa) Slide 27: H.E.R.E. step 2 with screenshot (continued)



Step 2: Assign T.H.E.S.E. Categories

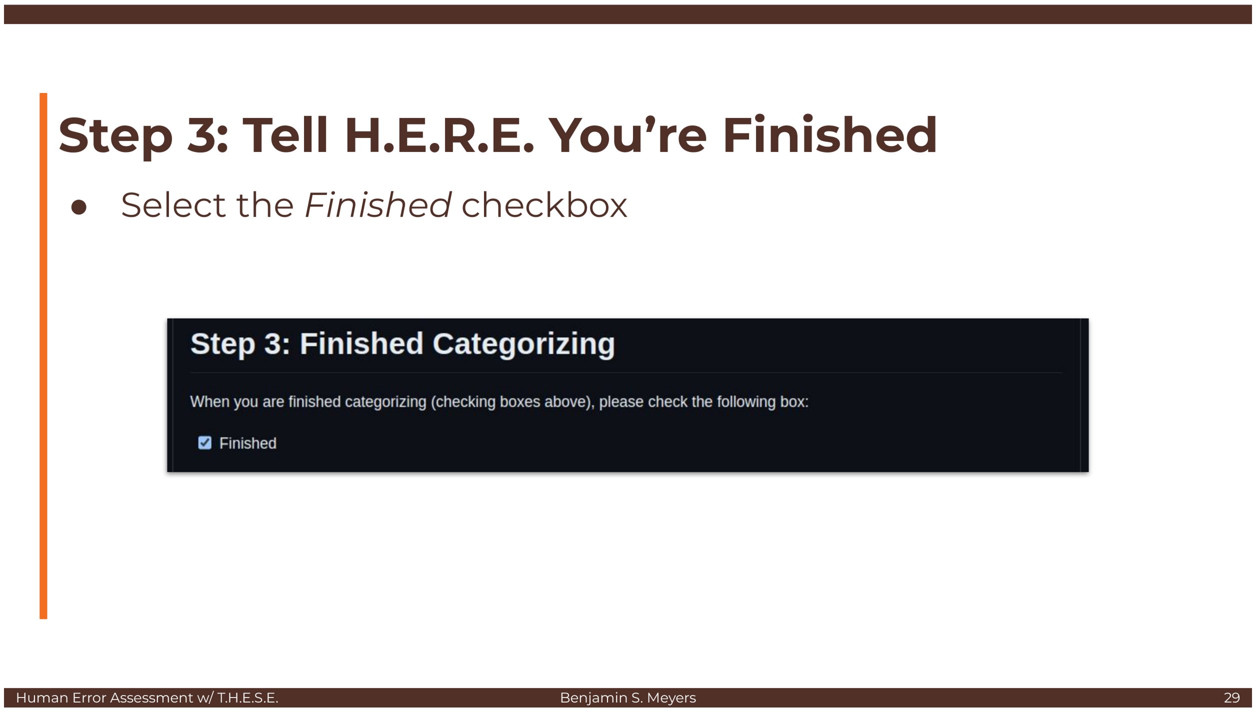
- H.E.R.E. applies relevant labels

github-actions bot added Overlooking Documented Information (HE S03) Slip (Failure of Attention) labels last week

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 28

(ab) Slide 28: H.E.R.E. step 2 response with screenshot

Figure H.1: Optional Follow-Up Survey Slides (Continued)



Step 3: Tell H.E.R.E. You're Finished

- Select the *Finished* checkbox

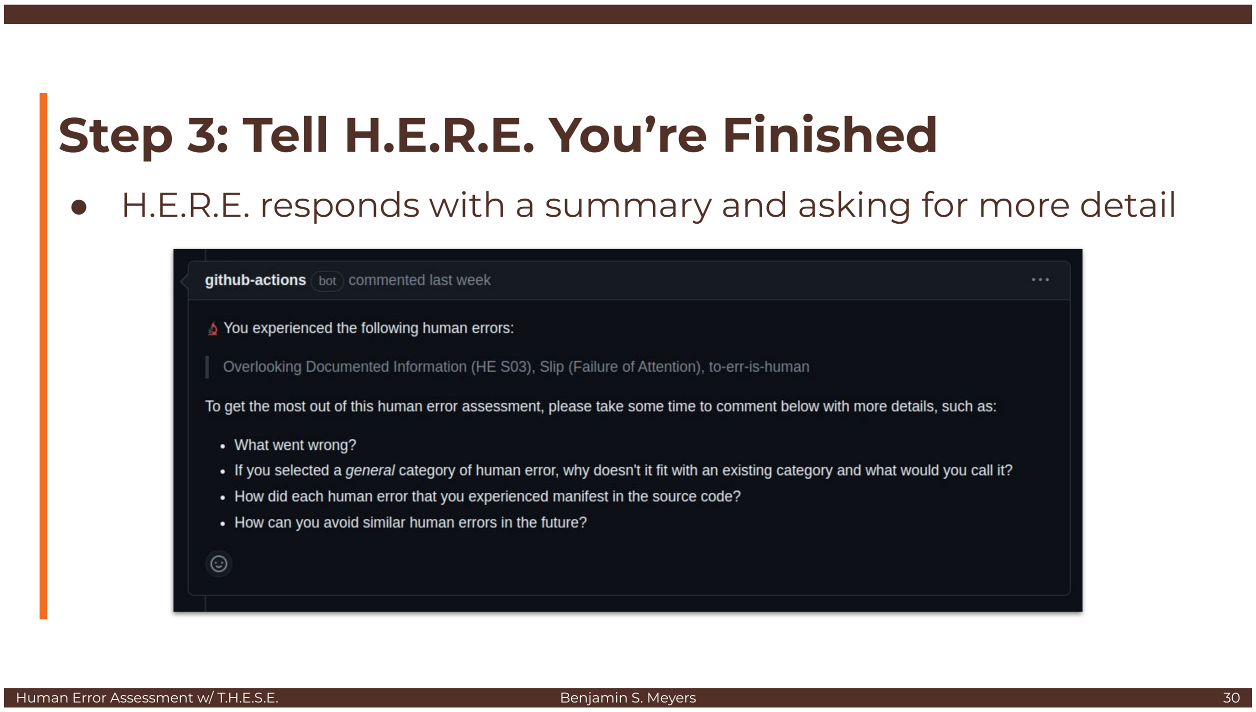
Step 3: Finished Categorizing

When you are finished categorizing (checking boxes above), please check the following box:

[✓ Finished]

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 29

(ac) Slide 29: H.E.R.E. step 3 with screenshot



Step 3: Tell H.E.R.E. You're Finished

- H.E.R.E. responds with a summary and asking for more detail

github-actions bot commented last week

You experienced the following human errors:

> Overlooking Documented Information (HE S03), Slip (Failure of Attention), to-err-is-human

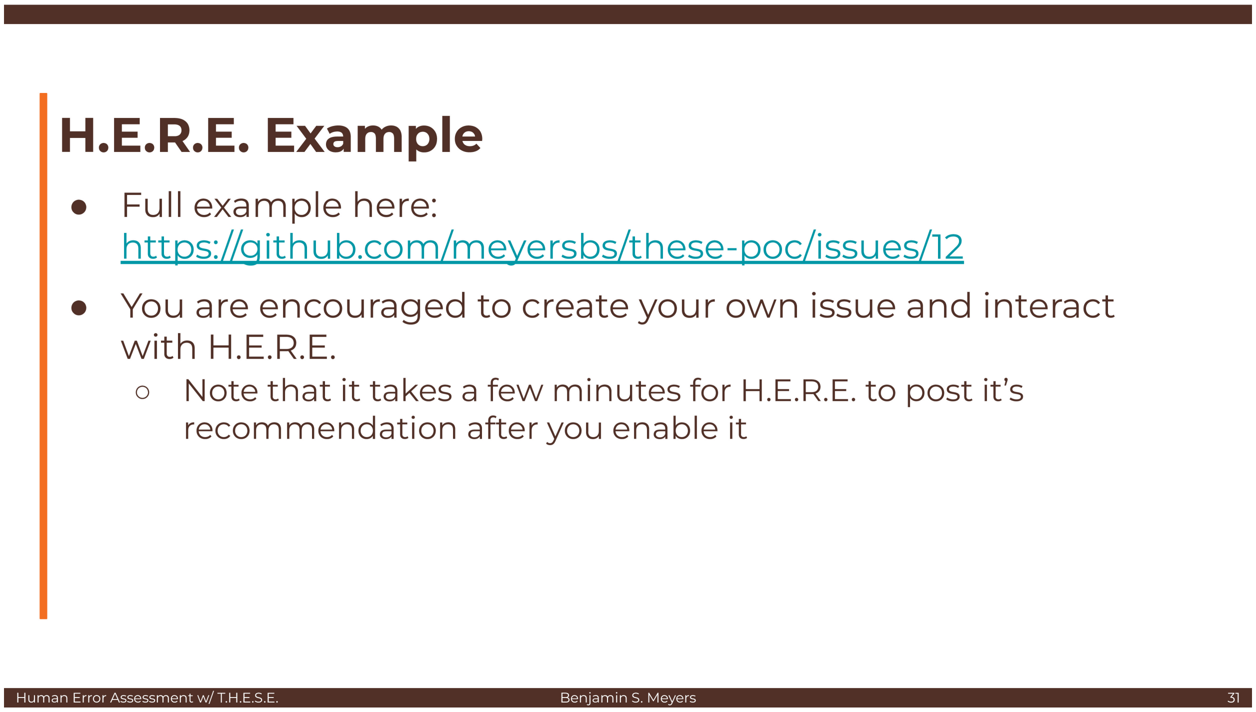
To get the most out of this human error assessment, please take some time to comment below with more details, such as:

- What went wrong?
- If you selected a *general* category of human error, why doesn't it fit with an existing category and what would you call it?
- How did each human error that you experienced manifest in the source code?
- How can you avoid similar human errors in the future?

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 30

(ad) Slide 30: H.E.R.E. step 3 response with screenshot

Figure H.1: Optional Follow-Up Survey Slides (Continued)

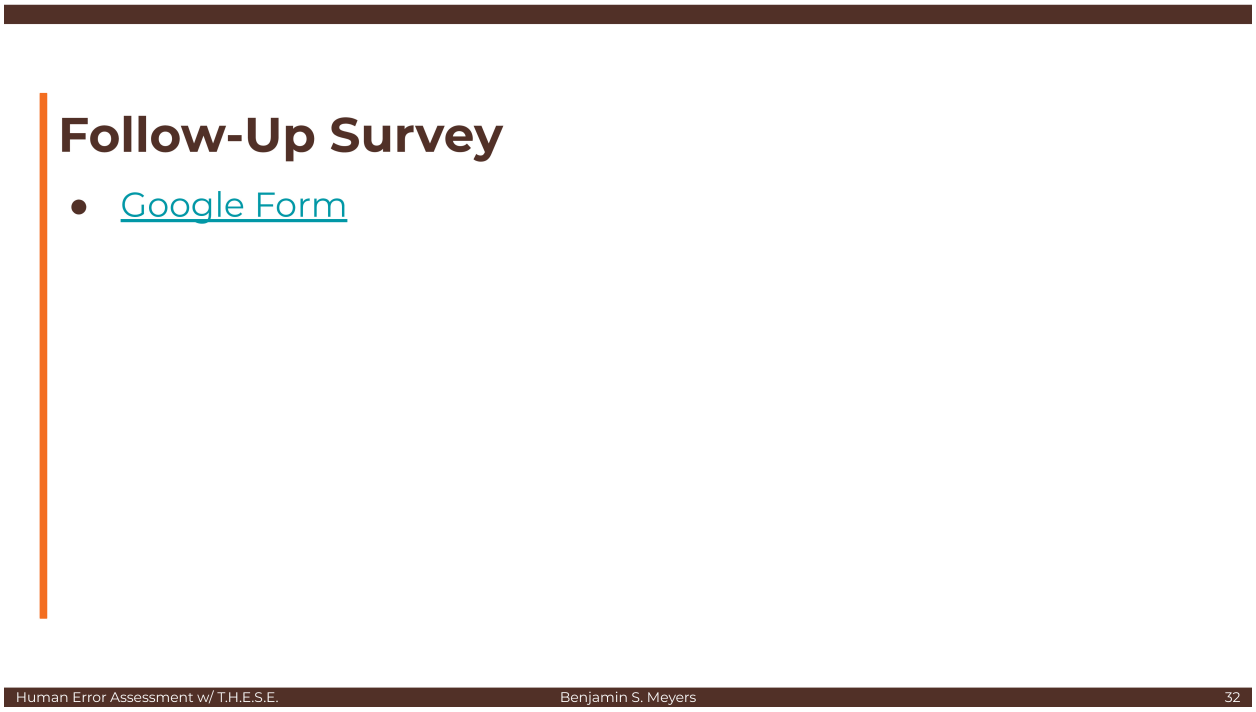


H.E.R.E. Example

- Full example here: https://github.com/meyersbs/these-poc/issues/12
- You are encouraged to create your own issue and interact with H.E.R.E.
  - Note that it takes a few minutes for H.E.R.E. to post it's recommendation after you enable it

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 31

(ae) Slide 31: Link to full example for H.E.R.E.



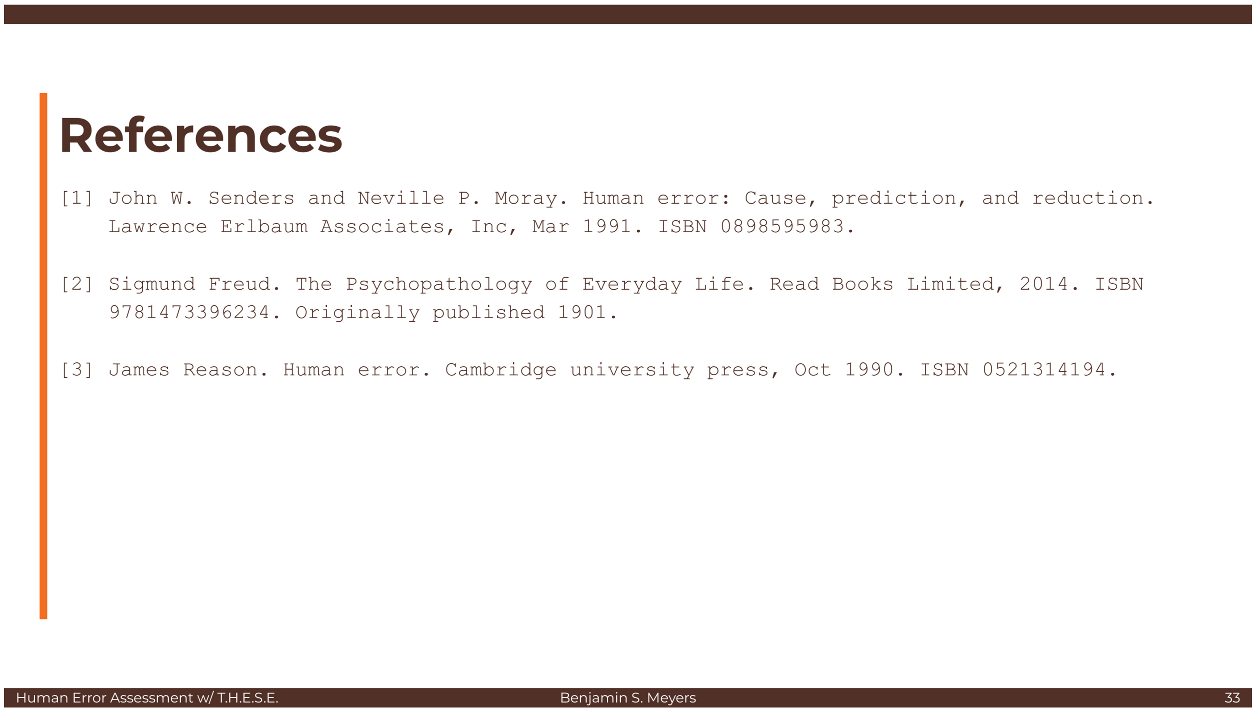
Follow-Up Survey

- Google Form

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 32

(af) Slide 32: Link to follow-up survey

Figure H.1: Optional Follow-Up Survey Slides (Continued)



References

[1] John W. Senders and Neville P. Moray. Human error: Cause, prediction, and reduction. Lawrence Erlbaum Associates, Inc, Mar 1991. ISBN 0898595983.

[2] Sigmund Freud. The Psychopathology of Everyday Life. Read Books Limited, 2014. ISBN 9781473396234. Originally published 1901.

[3] James Reason. Human error. Cambridge university press, Oct 1990. ISBN 0521314194.

Human Error Assessment w/ T.H.E.S.E. Benjamin S. Meyers 33

(ag) Slide 33: References

Figure H.1: Optional Follow-Up Survey Slides (Continued)

Appendix I

Follow-Up Survey

After reviewing the slides shown in Appendix H, user study participants were asked to complete the follow-up survey outlined in this appendix.

1. Which research study did you participate in?
2. Since completing participation in the research study...
   (a) I have continued to think about my software faults/failures in terms of slips, lapses, and mistakes.
   (b) I have continued to think about my software faults/failures in terms of T.H.E.S.E. categories.
   (c) I find T.H.E.S.E. valuable in my software engineering activities.
   (d) I find human error reflection valuable in my software engineering activities.
3. Please explain your answer to the previous question. Please be specific.
4. This human error reflection process using T.H.E.S.E. ...
   (a) would lead to meaningful reflection on my human errors.
   (b) would be a beneficial process for individual software engineers.
   (c) would be a beneficial process for software engineering teams.
   (d) has clear instructions.
   (e) is general enough to apply to all software engineering phases.
   (f) is confusing.
   (g) is overwhelming.
5. You may use this space to elaborate on any of your answers to the previous question.
6. What value do you see this human error reflection process providing to software engineers? Please be specific.
7. Do you have any recommendations for improving the human error reflection process using T.H.E.S.E.?
8. H.E.R.E. ...
   (a) would make it easy for software engineers to adopt human error reflection with T.H.E.S.E.
   (b) has a clear process.
   (c) is confusing.
   (d) is overwhelming.
9. Do you have any recommendations for improving the Human Error Reflection Engine (H.E.R.E.)?
10. Do you have any additional comments, concerns, or suggestions that you would like to share about T.H.E.S.E., human error reflection, or H.E.R.E.?

Figure I.1: Optional Follow-Up Survey Questions

7 months after the user study concluded, participants were sent this optional follow-up survey. Questions 2, 4, and 8 were answered on a five-point Likert scale (strongly agree, somewhat agree, neither agree nor disagree, somewhat disagree, strongly disagree). Question 1 was multiple choice with three potential responses: (A) April-May, 2022, (B) September-December, 2022, and (3) I don't remember. All other questions were open ended.

Appendix J

Follow-Up Responses

This appendix includes responses to the follow-up survey questions shown in Appendix I.

Table J.1: Optional Follow-Up Survey Responses

Q#	**Respondent/Response**
(1)	**A**: September-December, 2022
(2-a)	**A**: Strongly Agree
(2-b)	**A**: Strongly Agree
(2-c)	**A**: Strongly Agree
(2-d)	**A**: Strongly Agree
(3)	**A**: As a developer, I grow form my mistakes. Whether or not I'm aware of them, it's always important to make sure they don't happen (or don't happen again)
(4-a)	**A**: Strongly Agree
(4-b)	**A**: Strongly Agree
(4-c)	**A**: Strongly Agree
(4-d)	**A**: Strongly Agree
(4-e)	**A**: Strongly Agree
(4-f)	**A**: Somewhat Disagree
(4-g)	**A**: Strongly Disagree
(5)	**A**: N/A
(6)	**A**: Value of improvement, either individual or as a team
(7)	**A**: Maybe include mistakes outside of the knowledgeable scope of a software engineer. For example, there are some details that only a sponsor or product owner may know, so their mistakes can be just as critical as mistakes that engineers make.
(8-a)	**A**: Strongly Agree
(8-b)	**A**: Strongly Agree
(8-c)	**A**: Somewhat Disagree
(8-d)	**A**: Strongly Disagree
(9)	**A**: N/A
(10)	**A**: N/A

Appendix K

Human Error Countermeasures

Peters & Peters [285] suggest 26 general human error countermeasures, which we list below with relevant examples:

- **Single-error Tolerance**: Products, systems, machines, equipment should tolerate simple human errors without creating dangerous situations.
- **The Rule of Two**: Human errors do not exist in a vacuum; "designers should consider all of the risk factors that could increase the propensity or likelihood of human error [285]."
- **Interposition**: Put up barriers or shields to prevent common, unavoidable errors, *e.g.* syntax checking in IDEs and compilers, James Reason's Swiss Cheese Model [300] (*i.e.* defense-in-depth).
- **Sequestration**: Isolate human errors to prevent wide-ranging harm, *e.g.* sandboxed development environments.
- **Interlocks & Lockouts**: Implement access controls to protect against blind spots, *e.g.* access control lists, filesystem permissions.
- **Channelization**: Implement guides to channel human behavior, *e.g.* traffic lights, road lines, well-defined requirements.
- **Guides & Stops**: Implement physical guides to prevent harm, *e.g.* electrical sensors for table saws.
- **Automation**: Take the human out of the loop, *e.g.* unit tests.
- **Instructions**: Provide detailed, specific, and clear instructions, *e.g.* documentation, troubleshooting guides.
- **Training**: Provide training for software, tools, machines, and processes, *e.g.* domain-specific training, security certifications, human error training.
- **Behavior Modification**: "Interventions to improve job satisfaction, interpersonal relationships, and company cultures [285]," *e.g.* identifying systemic human errors and altering the software development process accordingly.
- **Safety Factors**: Define and measure metrics for risk, *e.g.* strength-vs-load ratios in material science, number-of-errors-occurring versus number-of-expected-errors.
- **Warnings**: Provide warnings for dangerous equipment, processes, software, and tools, *e.g.* code comments for complex functions, warnings before root-level operations.
- **Protective Equipment**: Provide protective equipment to minimize harm, *e.g.* welding masks, compilers, IDE tools.
- **Redundancy**: "The design should provide more than one means to accomplish a task [285]," *e.g.* data backups, redundant servers, multiple routes to host, pair-programming.

- **Derating**: Reducing the workload on the human, *e.g.* having the right number of software engineers assigned to a project.
- **Fail-safes**: Human errors leading to equipment failure should not create unacceptable risk, *e.g.* dead-man switches.
- **Stress Reduction**: Reduce psychological stress, anxiety, and fatigue, *e.g.* regular breaks, controlled task-switching (not multitasking).
- **Tools**: Provide the right tools for the job to reduce the potential of human error, *e.g.* risk assessment, debuggers, syntax checkers, unit tests.
- **Replacement**: "If all attempted countermeasures prove unequal to the task, at some point in time the person must be replaced by job reassignment or termination [285]." We caution against blaming human errors on individual software engineers. Instead of replacing people, we recommend having planned maintenance for software and hardware.
- **Enhancement**: Modifications to the human operator, *e.g.* night vision glasses, feedback on brake pedals.
- **Inactivity**: Inactive systems should not be able to cause harm, *e.g.* full shutdown.
- **Independent Confirmation**: Independently confirm unexpected and/or serious reports.
- **Therapy**: If the human is irreplaceable, but personal matters are affecting their work, therapy may be necessary.
- **Improvisation**: "There are situations in which extemporaneous remedies are fashioned, at the scene, to correct ongoing human errors [285]," *e.g.* reacting to quality assurance issues in real-time.
- **Pragmatism**: Behavior based human error classification with descriptive categories.

Appendix L

T.H.E.S.E. Changelog

Throughout this dissertation, T.H.E.S.E. went through many iterations and saw multiple changes, which we summarize in this appendix for quick reference. Note that extended descriptions were added (see Section 5.3.2.4) and are documented in Appendix G instead of below.

- **S01: Typos & Misspellings**
  - **Version 1:** N/A
  - **Version 2:** Typos and misspellings may occur in code comments, or when typing the name of a variable, function, or class.
  - **Version 3:** Typos and misspellings may occur in code comments, documentation (and other development artifacts), or when typing the name of a variable, function, or class. Examples include misspelling a variable name, writing down the wrong number/name/word during requirements elicitation, referencing the wrong function in a code comment, and inconsistent whitespace (that does not result in a syntax error).
  - **Version 4:** N/A
- **S02: Syntax Errors**
  - **Version 1:** Any error in coding language syntax that impacts the executability of the code. Note that Logical Errors (*e.g.* `+=` instead of `+`) are not Syntax Errors.
  - **Version 2:** N/A
  - **Version 3:** Any error in coding language syntax that impacts the executability of the code. Note that Logical Errors (*e.g.* `+=` instead of `+`) are not Syntax Errors. Examples include mixing tabs and spaces (*e.g.* Python), unmatched brackets/braces/parenthesis/quotes, and missing semicolons (*e.g.* Java).
  - **Version 4:** N/A
- **S03: Overlooking Documented Information**
  - **Version 1:** Errors resulting from overlooking documented information, such as project descriptions, stakeholder requirements, API/library/tool/framework documentation, coding standards, programming language specifications, and bug/issue reports.
  - **Version 2:** N/A
  - **Version 3:** Errors resulting from overlooking (internally and externally) documented information, such as project descriptions, stakeholder requirements, API/library/tool/framework documentation, coding standards, programming language specifications, bug/issue reports, and looking at the wrong version of documentation or documentation for the wrong project/software.
  - **Version 4:** N/A

- **S04: Multitasking Errors**
  - **Version 1:** N/A
  - **Version 2:** Errors resulting from multitasking.
  - **Version 3:** N/A
  - **Version 4:** Errors resulting from multitasking, *i.e.* working on multiple software engineering tasks at the same time.
- **S05: Hardware Interaction Errors**
  - **Version 1:** N/A
  - **Version 2:** Attention failures while using computer peripherals, such as mice, keyboard, and cables. Examples include copy/paste errors, clicking the wrong button, using the wrong keyboard shortcut, and incorrectly plugging in cables.
  - **Version 3:** N/A
  - **Version 4:** N/A
- **S06: Overlooking Proposed Code Changes**
  - **Version 1:** N/A
  - **Version 2:** Errors resulting from lack of attention during formal/informal code review.
  - **Version 3:** N/A
  - **Version 4:** Errors resulting from lack of attention during formal/informal code review. Examples include overlooking incorrect logic, or skipping files, functions, or classes during a review.
- **S07: Overlooking Existing Functionality**
  - **Version 1:** N/A
  - **Version 2:** N/A
  - **Version 3:** Errors resulting from overlooking existing functionality, such as reimplementing variables, functions, and classes that already exist, or reimplementing functionality that already exists in a standard library.
  - **Version 4:** Errors resulting from overlooking existing functionality, such as reimplementing or duplicating variables, functions, and classes that already exist, or reimplementing functionality that already exists in a standard library. Other examples include deleting necessary variables, functions, and classes.
- **S08: General Attentional Failure**
  - **Version 1:** N/A
  - **Version 2:** Only use this category if you believe your error to be the result of a lack of attention, but no other slip category fits.
  - **Version 3:** N/A
  - **Version 4:** N/A
- **L01: Forgetting to Finish a Development Task**
  - **Version 1:** N/A
  - **Version 2:** Forgetting to implement a required feature.
  - **Version 3:** Forgetting to finish a development task. Examples include forgetting to implement a required feature, forgetting to finish a user story, and forgetting to deploy a security patch.
  - **Version 4:** N/A

- **L02: Forgetting to Fix a Defect**
  - **Version 1:** Forgetting to fix a defect that you encountered, but chose not to fix right away.
  - **Version 2:** N/A
  - **Version 3:** N/A
  - **Version 4:** N/A
- **L03: Forgetting to Remove Development Artifacts**
  - **Version 1:** Forgetting to remove debug log files, dead code, informal test code, commented out code, test databases, backdoors, *etc.*
  - **Version 2:** N/A
  - **Version 3:** Forgetting to remove debug log files, dead code, informal test code, commented out code, test databases, backdoors, *etc.* Examples include leaving unnecessary code in the comments, and leaving notes in internal development documentation.
  - **Version 4:** N/A
- **L04: Working with Outdated Source Code**
  - **Version 1:** N/A
  - **Version 2:** Forgetting to git-pull (or equivalent in other version control systems), or using an outdated version of a library.
  - **Version 3:** N/A
  - **Version 4:** N/A
- **L05: Forgetting an Import Statement**
  - **Version 1:** N/A
  - **Version 2:** Forgetting to import a necessary library, class, variable, or function, or forgetting to include arguments in a function call.
  - **Version 3:** N/A
  - **Version 4:** Forgetting to import a necessary library, class, variable, or function, or forgetting to access a property, attribute, or argument. Examples include forgetting to import python's `sys` library, forgetting to include a header file in C, or forgetting to pass an argument to a function.
- **L06: Forgetting to Save Work**
  - **Version 1:** Forgetting to push code, or forgetting to backup/save data or documentation.
  - **Version 2:** N/A
  - **Version 3:** N/A
  - **Version 4:** N/A
- **L07: Forgetting Previous Development Discussion**
  - **Version 1:** N/A
  - **Version 2:** Errors resulting from forgetting details from previous development discussions.
  - **Version 3:** N/A
  - **Version 4:** N/A

- **L08: General Attentional Failure**
  - **Version 1:** N/A
  - **Version 2:** Only use this category if you believe your error to be the result of a memory failure, but no other lapse category fits.
  - **Version 3:** N/A
  - **Version 4:** N/A
- **M01: Code Logic Errors**
  - **Version 1:** A code logic error is one in which the code executes, but produces an incorrect output/behavior due to incorrect logic. Examples include using incorrect operators (*e.g.* . `+=` instead of `+`), erroneous if/else statements, incorrect variable initializations, problems with variable scope, and omission of necessary logic.
  - **Version 2:** N/A
  - **Version 3:** A code logic error is one in which the code executes (*i.e.* actually runs), but produces an incorrect output/behavior due to incorrect logic. Examples include using incorrect operators (*e.g.* `+=` instead of `+`), erroneous if/else statements, incorrect variable initializations, problems with variable scope, and omission of necessary logic.
  - **Version 4:** N/A
- **M02: Incomplete Domain Knowledge**
  - **Version 1:** Errors resulting from incomplete knowledge of the software system's target domain (*e.g.* banking, astrophysics).
  - **Version 2:** N/A
  - **Version 3:** N/A
  - **Version 4:** Errors resulting from incomplete knowledge of the software system's target domain (*e.g.* banking, astrophysics). Examples include planning/designing a system without understanding the nuances of the domain.
- **M03: Wrong Assumption Errors**
  - **Version 1:** Errors resulting from an incorrect assumption about system requirements, stakeholder expectations, project environments (*e.g.* coding languages and frameworks), library functionality, and program inputs.
  - **Version 2:** N/A
  - **Version 3:** N/A
  - **Version 4:** N/A
- **M04: Internal Communication Errors**
  - **Version 1:** Errors resulting from inadequate communication between development team members.
  - **Version 2:** N/A
  - **Version 3:** N/A
  - **Version 4:** Errors resulting from inadequate communication between development team members. Examples include misunderstanding development discussion, misinterpreting or providing ambiguous instructions, communicating using the wrong medium (*e.g.* oral vs. written), or communicating ineffectively (*e.g.* too formal/informal, too much unnecessarily complex language, hostile language/body language).

- **M05: External Communication Errors**
  - **Version 1:** Errors resulting from inadequate communication with project stakeholders, third-party contractors, or users.
  - **Version 2:** N/A
  - **Version 3:** N/A
  - **Version 4:** Errors resulting from inadequate communication with project stakeholders or third-party contractors. Examples include providing ambiguous or unclear directions to third-parties or users, or misinterpreting stakeholder feedback, communicating using the wrong medium (*e.g.* oral vs. written), or communicating ineffectively (*e.g.* too formal/informal, too much unnecessarily complex language, hostile language/body language).
- **M06: Solution Choice Errors**
  - **Version 1:** Misunderstood problem-solving methods/techniques result in analyzing the problem incorrectly and choosing the wrong solution. For example, choosing to implement a database system in Python rather than using SQL.
  - **Version 2:** N/A
  - **Version 3:** N/A
  - **Version 4:** Misunderstood problem-solving methods/techniques result in analyzing the problem incorrectly and choosing the wrong solution. For example, choosing to implement a database system in Python rather than using SQL, or choosing the wrong software design pattern. Overconfidence in a solution choice also falls under this category.
- **M07: Time Management Errors**
  - **Version 1:** N/A
  - **Version 2:** Errors resulting from a lack of time management, such as failing to allocate enough time for the implementation of a feature.
  - **Version 3:** N/A
  - **Version 4:** Errors resulting from a lack of time management, such as failing to allocate enough time for the implementation of a feature, procrastinating a development task, or predicting the time required for a task incorrectly.
- **M08: Inadequate Testing**
  - **Version 1:** N/A
  - **Version 2:** Failure to implement necessary test cases, failure to consider necessary test inputs, or failure to implement a certain type of testing (*e.g.* unit, penetration, integration) when it is necessary.
  - **Version 3:** N/A
  - **Version 4:** Failure to implement necessary test cases, failure to consider necessary test inputs, failure to implement a certain type of testing (*e.g.* unit, penetration, integration) when it is necessary, or failure to consider edge cases or unexpected inputs.
- **M09: Incorrect/Insufficient Configuration**
  - **Version 1:** N/A
  - **Version 2:** Errors in configuration of libraries/frameworks/environments or errors related to missing configuration options.
  - **Version 3:** Errors in configuration of libraries/frameworks/environments or errors related to missing configuration options. Examples include misconfigured IDEs, improper directory structure for a specific programming language, and missing SSH keys.
  - **Version 4:** Errors in configuration of libraries/frameworks/environments or errors related to missing configuration options. Examples include misconfigured IDEs or text editors, improper directory structure for a specific programming language, missing SSH keys, missing or incorrectly named database fields or tables, missing or incorrectly named/formatted configuration files, or not installing a required library.

- **M10: Code Complexity Errors**
  - **Version 1:** Errors resulting from misunderstood code due to poor documentation or unnecessary complexity. Examples include too many nested if/else statements or for-loops and poorly named variables/-functions/classes/files.
  - **Version 2:** N/A
  - **Version 3:** N/A
  - **Version 4:** N/A
- **M11: Internationalization/String Encoding Errors**
  - **Version 1:** N/A
  - **Version 2:** Errors related to internationalization and/or string/character encoding.
  - **Version 3:** N/A
  - **Version 4:** Errors related to internationalization and/or string/character encoding. Examples include using ASCII instead of Unicode, using UTF8 when UTF16 was necessary, failure to design the system with internationalization in mind, or failing to verify the character length of user input.
- **M12: Inadequate Experience Errors**
  - **Version 1:** N/A
  - **Version 2:** Errors resulting from inadequate experience with a language, library, framework, or tool.
  - **Version 3:** N/A
  - **Version 4:** N/A
- **M13: Insufficient Tooling Access Errors**
  - **Version 1:** N/A
  - **Version 2:** Errors resulting from not having sufficient access to necessary tooling. Examples include not having access to a specific operating system, library, framework, hardware device, or not having the necessary permissions to complete a development task.
  - **Version 3:** N/A
  - **Version 4:** N/A
- **M14: Workflow Order Errors**
  - **Version 1:** N/A
  - **Version 2:** Errors resulting from working out of order, such as implementing dependent features in the wrong order, implementing code before the design is stabilized, releasing code that is not ready to be released, or skipping a workflow step.
  - **Version 3:** N/A
  - **Version 4:** N/A
- **M15: General Planning Failure**
  - **Version 1:** N/A
  - **Version 2:** Only use this category if you believe your error to be the result of a planning failure, but no other mistake category fits.
  - **Version 3:** N/A
  - **Version 4:** N/A

Appendix M

Selected Quotes

This appendix lists quotes from references that we would like to share.

> *"Software rarely works as intended... 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*** *[191].*

> *"Human errors are the primary cause of software defects, since computer programs are a pure cognitive product that describes its designers' thoughts.* ***Understanding the human error mechanisms of software developers will advance*** *various approaches that are currently used to defend against software defects, such as defect prevention, defect prediction, defect detection and fault tolerance [142]."*

> *"**Software errors are human errors**, software is written by man, and indeed man does make mistakes... errors are always present. Some errors are more harmful, visible or costly than others, and testing may never be able to reveal all of them. It seems that software errors cannot be totally prevented, so the best we can do is to try to locate them as early as possible, and at least find and fix the most harmful ones [198]."*

> *"**All of us have experienced human error.** When we interact with machines or complex systems, we frequently do things that are contrary to our intentions. Depending on the complexity of the system and the intentions of the people interacting with it, this can be anything from an inconvenience (often it is not even noticed) to a genuine catastrophe. Human error can occur in the design, operation, management, and maintenance of the complex systems characteristic of modern life. Because we depend increasingly on these systems for our well being, it is clear that human error is a potent and frequent Link of hazard to human life and welfare, and to the ecosystems of Earth [320]."*

> *"**People are key components of processes**. They are involved in process design, operation, maintenance, etc. No step in the process life cycle is without some human involvement. Based on human nature, **human error is a given and will arise in all parts of the process life cycle**. Also, processes are generally not well-protected from human errors since many safeguards are focused on equipment failure. Consequently, it is likely that human error will be an important contributor to risk for most processes [35]."*

> *"**Human error continues to be a major computer security issue**, although many contemporary information security practitioners appear to have forgotten about it [362]."*

> *"Targeting the 'human' part of the perceived 'human error' problem is still quite popular. As in: **keep beating the human until the error goes away**. But your 'human error' problem is in all likelihood more about the organization than it is about the human [76]."*

> *"**To Err Is Human** asserts that the problem is not bad people... it is that good people are working in bad systems that need to be made safer [83]."*

> *"If the HR department is involved in safety in your organization, then an incident investigation can quickly degenerate from learning opportunity to performance review [76]."*

"Because of the complexity of human behavior and the characteristics of human errors, ***it is impossible to eliminate all the human errors****. However, these human errors could and should be reduced to the utmost. One of the effective ways is to study the root causes of the human errors and then propose the improvement measure and prevent these errors [367]."*

"Unfortunately, ***modern system designs do not take into account the possibility of human error****. Traditional high-end fault-tolerant systems have a partial solution in that their vendors lock up their systems and give the keys only to certified, trained service personnel. But even a highly-trained operator will inevitably make mistakes, so this is hardly a complete solution. Furthermore, it is a solution that does not apply to modern Internet service environments, where systems consist of collections of hardware and software from different vendors deployed in highly varied configurations [51]."*

"Therefore, ***instead of blaming the human who happens to be involved****, it would be better to try to identify the system characteristics that led to the incident and then to modify the design either by elimination of the situation or at least minimization of the impact for future events. One major step would be to remove the term "human error" from our vocabulary and to re-evaluate the need to blame individuals. A second major step would be to develop design specifications that consider the functionality of the human with the same degree of care that has been given to the rest of the system [262]."*

"On an individual level, one reason for our dislike of errors is that they cause us distress. ***Errors show our deficiencies****, including where we did not pay enough attention, or when we misjudged a situation, thus questioning our reputation and our pride as proficient workers. Besides, errors may be dangerous and can cause adverse things to happen. On the organizational level, errors can endanger the creation of economic value and may also put employees, clients, or customers at risk. The research on safety and accidents has endless examples of minor errors leading to disastrous outcomes [34]."*

*"****There is however a positive face of mistakes****, essential in learning, in teaching, in scientific research and in any creative work [199]."*

"On the one hand, professionals as well as companies are keen to avoid errors; on the other hand, scholars have indicated that errors cannot be completely prevented and that a heavy reliance on error prevention can have detrimental effects. Instances of such detrimental effects are: the potential occurrence of errors may be insufficiently anticipated; employees lose their skills in dealing with them; and learning opportunities are missed. For these reasons, a shift from an exclusive error prevention approach to an error management strategy has been proposed. Error management concepts suggest, in addition to prevention, an efficient way of dealing with errors and learning from them. The error management approach is based on the assumption that a ***systematic analysis of occurring errors can provide organisations with information about necessary adjustments of knowledge, strategies, and behaviour****. Moreover,* ***errors may evoke new insights that lead to learning beyond the mere prevention of similar errors****. Hence, although it seems obvious that errors should be avoided in professional work because they endanger the attainment of desired goals, a prerequisite for avoiding errors as well as for capturing the potential benefits that arise through errors is to be open to their occurrence and to learn from them [34]."*

"Knowledge about possible errors can play an important role when making up plans about how to solve a task at hand... similarly... the ability of employees to anticipate errors is an important cornerstone of their performance. ***If errors are anticipated, they may be avoided entirely or better coped with*** *when they do occur [34]."*

*"****When you go behind the label 'human error,' you see people and organizations trying to cope with complexity****, continually adapting, evolving along with the changing nature of risk in their operations. Such coping with complexity, however, is not easy to see when we make only brief forays into intricate worlds of practice. Particularly when we wield tools to count and tabulate errors, with the aim to declare war on them and make them go away, we all but obliterate the interesting data that is out there for us to discover and learn how the system actually functions.* ***As practitioners confront different evolving situations, they navigate and negotiate the messy details of their practice to bridge gaps and to join together the bits and pieces of their system, creating success as a balance between the multiple conflicting goals and pressures imposed by their organizations.*** *In fact, operators generally do this job so well, that the adaptations and effort glide out of view for outsiders and insiders alike. The only residue left, shimmering on the surface, are the 'errors' and incidents to be fished out by those who conduct short, shallow encounters in the form of, for example, safety audits or error counts.* ***Shallow encounters miss how learning and adaptation are ongoing****—without these, safety cannot even be maintained in a dynamic and changing organizational setting and environment—yet these adaptations lie mostly out of immediate view, behind labels like 'human error [363].''*

Appendix N

Behind Human Error

In *Behind Human Error* [363], the authors argue that *human error* is not a problem, but a misleading (and perhaps outdated) label, behind which exists a vast ecosystem of organizational problems and opportunities for learning:

> *"If you think you have a 'human error' problem, don't think for a minute that you have said anything meaningful about the causes of your troubles, or that a better definition or taxonomy will finally help you get a better grasp of the problem, because you are looking in the wrong place, and starting from the wrong position. You don't have a problem with erratic, unreliable operators. You have an organizational problem, a technological one. You have to go* ***behind*** *the label human error to begin the process of learning, of improvement, of investing in safety. The 10 steps forward summarize general patterns about error and expertise, complexity, and learning. These 10 steps constitute a checklist for constructive responses when you see a window of opportunity to improve safety. Here they are:"*
>
> 1. *"Recognize that human error is an attribution."*
> 2. *"Pursue second stories to find deeper, multiple contributors to failure."*
> 3. *"Escape the hindsight bias."*
> 4. *"Understand work as performed at the sharp end of the system."*
> 5. *"Search for systemic vulnerabilities."*
> 6. *"Study how practice creates safety."*
> 7. *"Search for underlying patterns."*
> 8. *"Examine how change will produce new vulnerabilities and paths to failure."*
> 9. *"Use new technology to support and enhance human expertise."*
> 10. *"Tame complexity."*

While my background is not philosophy, philosophical questions keep me up at night. The study of human error breeds philosophical questions, and the deeper you dig, the more questions you encounter, and the harder you think. Here, I examine my dissertation under the lens of going **behind** human error, with the goal of raising philosophical questions for the reader:

1. *"Recognize that human error is an attribution."* — Attributing the cause of a failure to human error results in a concise post-mortem (*i.e.* a *first story* [363]), but it shields us from further knowledge and understanding of the failure and its causes. To facilitate deeper understanding, our human error reflection process requires software engineers confronting their human errors to go beyond T.H.E.S.E. categorization and consider questions about how the resulting defect came to be, how the human error is related to others, and mitigation strategies.

2. *"Pursue second stories to find deeper, multiple contributors to failure."* — The *first stories* from literature and GitHub artifacts that we aggregated into T.H.E.S.E. provide a tool for software engineers, but T.H.E.S.E. categorization should not be considered the resolution of a human error informed micro post-mortem. During our user study (Section 5.2), we elicited many *second stories*, which revealed a wealth of nuance in human error categorization and reflection. T.H.E.S.E. is a shared vocabulary for software engineering teams to discuss their second stories.
3. *"Escape the hindsight bias."* — When we examine a software failures after-the-fact, we are aware of the outcome, and our conclusions can be biased. Unless we personally experienced the human error(s) behind the software failure in question, the true details are hidden from us. Our goal, restated, is ***to help software engineers confront and reflect on their human errors by creating a process to document, organize, and analyze human errors***. Our human error reflection process is a personal process intended to suss out the messy details of the human error and wash away first stories by bringing second stories to the surface.
4. *"Understand work as performed at the sharp end of the system."* — At the *blunt end*, the project stakeholders and the organization place constraints on software engineering. Those at the blunt end are concerned primarily with the final software product, while the software engineers working at the *sharp end* have to navigate the imposed constraints, balancing complexity, security, quality, and functionality. Those working at the blunt end may be tempted to blame software engineers for the human errors they experience, but this would be a *mistake*.
5. *"Search for systemic vulnerabilities."* — Safety, much like security, is an emergent property of systems [363]; safety must be created through a process of continual reflection on human errors followed by mitigation. T.H.E.S.E. provides a lens to examine recurrent human errors, which in turn allows systemic human errors to come to light.
6. *"Study how practice creates safety."* — Software engineers can never inherently know what the future will hold (if they could, software would be perfect and there would be no software defects). But we can examine the past in an effort to predict the future. Examining trends in CVE and CWE allows software engineers to focus their efforts on typical vulnerabilities and weaknesses. In that vain, examining trends in experienced human errors will enable software engineers to focus their attention on implementing the appropriate tools and processes to mitigate typical human errors.
7. *"Search for underlying patterns."* — Patterns help us reduce large sets of information into digestible bullet points, thus allowing us to make conclusions and decisions. We have done the hard work of aggregating patterns in software engineers' human errors—from 192 human errors in literature, 200 self-admitted human errors on GitHub, and 162 human errors experienced in our user study—into T.H.E.S.E., and we have designed our human error reflection process to empower software engineers to further identify human error patterns in their work.
8. *"Examine how change will produce new vulnerabilities and paths to failure."* — The computing field is continually changing as algorithms, languages, tools, and paradigms rise and fall. Human error reflection has to be a continuous process for software engineers, which, in part, prompted us to implement H.E.R.E., to allow for easy adoption and integration of human error reflection into existing software engineering workflows.
9. *"Use new technology to support and enhance human expertise."* — New technologies can be both an aid and a hindrance. While it could be argued that human error reflection with T.H.E.S.E. adds too much strain on the already complex software engineering process, we took care to minimize any potential negative impacts. We reduced ambiguity of T.H.E.S.E. categories while also verifying that T.H.E.S.E., our human error reflection process, and H.E.R.E. are easy to use, comprehensive, and valuable to software engineers.
10. *"Tame complexity."* — While the complexity of software development can never truly be tamed, it can be managed through continual human error reflection. In confronting, organizing, and reacting to their human errors, the software engineering community will be closer to inculcating the wisdom of historical developer human errors, enabling them to engineer higher quality and more secure software.

Appendix O

Human Error & Philosophy

Toward the end of my dissertation, I had the pleasure of attending Supercompting '22 with a coworker, Andy Elble. One day, after 12 hours of presentations, we decided to grab dinner and a few beers at our hotel. After brain-dumping what we learned during the day, and planning how we intended to incorporate those lessons and insights into our work, our discussion diverged. While this discussion is not immediately related to software engineering, it raises some interesting philosophical questions about human error, which, I feel, would be a shame not to publish. Here is a summary of the ideas we discussed:

Idea 1: *Error is the second law of thermodynamics intruding into our desire for order.*

The second law of thermodynamics establishes the concept of *entropy*—a state of disorder, randomness, or uncertainty. Entropy succeeds when a system fails, when human error forces the "system outside its acceptable limits [320]."

So, perhaps our goal shouldn't be to eliminate all human errors (which is unachievable), but instead to maximize the amount of time before entropy catches up. After all, there is no order without chaos.

Idea 2: *At large timescales, errors are just steps within the process.*

Consider natural selection, the process of countless genetic variations being tested and selected over massive timescales. Selected variations may be overwritten by new variations, or rejected variations may return based on environmental factors. Those rejected variations could be considered errors, but ultimately, they are part of the process.

Let's apply this concept to humans. We are constantly trying new things and exploring new ideas. We reject hypotheses that we deem too difficult to test and we forget old ideas, only to rediscover them later. The ancient world (and, unfortunately, some modern humans whom education has failed) believed that the Earth was flat. Through observation, we later learned that the Earth is spherical, and through measurement we came to know that the Earth is actually an ellipsoid. We used to believe that the Earth was the center of the Universe, and the Sun revolved around us. Then we learned that the Earth actually revolves around the Sun, but we still believed our solar system to be the center of the Universe. Now we know that our solar system is simply at the center of our *observable* Universe, and we ask ourselves what's outside our view.

These old ideas were human errors in logic, but they were important stepping stones in our pursuit of truth.

Idea 3: *Rules are the codified version of intent; error is the gap between intent and rules.*

Consider the three laws of robotics put forth by Isaac Asimov:

First Law:	A robot may not injure a human being or, through inaction, allow a human being to come to harm.
Second Law:	A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.
Third Law:	A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

These laws are a set of rules, a part of the robots' core programming, intended to prevent them from causing harm to humans. However, Asimov's robot stories frequently involve robots behaving in strange and/or counterintuitive ways as a result of their programming.

For example, in the story ***Liar!***, a telepathic robot named Herbie lies to a human named Susan about a coworker having romantic feelings for her. Herbie lied to avoid hurting Susan's feelings, following his First Law programming. However, Susan later finds out that her coworker does not love her, which hurts her more because Herbie's lie gave her hope. Upon receiving this new data—lying or telling the truth could have caused Susan pain (could have violated the First Law)—Herbie's *brain* locks up [24].

Herbie's error was lying to avoid breaking the First Law, but the intent of the First Law was to cause *minimal* harm to humans. This scenario is likely quite familiar to most human readers.

Idea 4: *A system that broadens the solution space broadens the opportunity for human error.*

Consider the card game *War*, where a standard deck of 52 cards is randomly shuffled and evenly divided between two players. Players take turns flipping the top card of their deck over; the player with the higher-valued card wins the round and takes both cards. If the cards flipped in a round are the same value, each player flips the next card from the top of their deck, and the player with the higher-valued card wins the round and takes all of the cards. The game ends when the winning player has taken all of the cards. Assuming no cheating occurs, there is zero opportunity for human error in the game of *War*, since there are no decisions to be made.

Now consider the game of *Tic-Tac-Toe*, which is played on a 3x3 grid of squares. The first player places an `X` in a square. The second player places an `O` in a different square. Players alternate placing `X`'s and `O`'s until one player has three of their markers in a row (vertically, horizontally, or diagonally), or the grid is full with no winning chain of markers. There are 19,683 different orientations of `X`'s and `O`'s that can be played—19,683 different solutions to the game [123]. The first player has to decide between 9 squares, the next player between 8 squares, then 7, and so on. If the game ends after a minimum of 5 turns, then the first player made 3 decisions, and the second player made 2 decisions. If the game ends after a maximum of 9 turns, then the first player made 5 decisions, and the second player made 4 decisions. With each decision, there is an opportunity for human error. Assuming any of the decisions can be erroneous, then there are up to 9 human errors in every game of *Tic-Tac-Toe*.

Now consider the game of *Chess*, played on an 8x8 grid of squares, where each player commands a King, Queen, two Bishops, two Knights, two Rooks, and eight Pawns. The goal of *Chess* is to force the opponent's King into a position where it cannot be moved. The mechanics of *Chess* are complicated and irrelevant to this discussion; all we need to know is that there are approximately 10^{43} possible solutions to *Chess* [324]. If the average game consists of 40 rounds (one move per player in each round), then there are at least 80 decisions to be made in the average game—at least 80 opportunities for human error.

The solution space for *Chess* is significantly higher than the solution space for *Tic-Tac-Toe*, and thus there is more opportunity for human error in a *Chess* game.

Let's consider another example: *complexity is the enemy of security* [204]. If we think about the attack surface of a software system—the available entry points for an attacker—as the solution space, then it is clear that as the complexity of the software system increases, so does its attack surface.

Appendix P

Meneely's Meditations

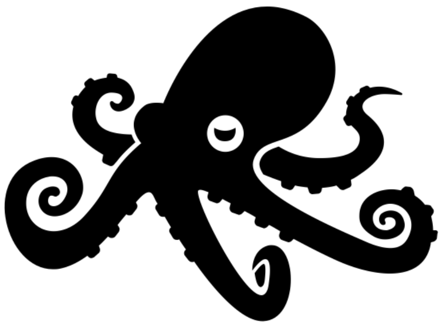
When my advisor graduated, he presented his advisor, Dr. Laurie Williams, with a list of *Laurie's Laws*—bits of wisdom that Dr. Meneely had gained from her throughout his dissertation. In keeping with that tradition, here I present *Meneely's Meditations*, a collection of wisdom and brain-churning comments from Andy, in no particular order:

1. The word *it* is a null pointer exception.
2. Papers are a mix of expository and persuasive writing.
3. Science makes for a terrible religion; it is always very unsure of itself.
4. Optimizations are rarely optimal.
5. Good writing is like a good user interface; it reduces the memory allocation required by the user.
6. Science is the worst thing a perfectionist can do.
7. Doing something that's never been done isn't necessarily a good idea; maybe it's never been done before because it's a bad idea.
8. If you never look at your data, you'll never understand your results.
9. Engineering is a series of decisions.
10. Paranoia is counterproductive.
11. It doesn't exist unless it's in a repository.
12. Clarity is king. I don't care about being informal as long as I'm both precise and clear.

Appendix Q

Concerning the Octopus

You may be wondering why you see a small image of an octopus acting as a dinkus (a paragraph separator) throughout this dissertation. You know, this little guy:



The octopus is a very misunderstood creature, often considered the closest thing to truly *"alien"* life on Earth. Indeed, if you just look at an octopus, it does feel alien with its ability to change colors and shapes at whim. The octopus even lives deep in the ocean, in the hydrosphere, far from the lithosphere and atmosphere that we call home.

But if you look beyond the surface, the octopus is a fascinating creature. With its brain distributed throughout its whole body, an octopus' arms can act independently without communicating with its central brain to make decisions. They are highly intelligent problem solvers, capable of escaping from closed jars, and solving mazes and other puzzles, and perhaps even communicating with each other. When they are being hunted, they typically choose to hide or evade their predators with clouds of ink, rather than attack [117]. We also still have a lot to learn about them; they are notoriously solitary creatures, yet we just recently discovered the fourth known octopus nursery—a small underwater *city* where octopuses congregate [289].

Human errors are much like octopuses. They hide behind labels such as "fault", "accident", or "mistake,' and their nuance is often locked within the minds of the humans who experience them. Outsiders examine human errors without the full picture and conclude that *'I never would have done that, why would you? It's so obvious.'* Human errors can also be scary, and confronting them can make us feel uncomfortable. But when we look beyond the surface, and reflect on our human errors, they can teach us about ourselves, and about each other.

This particular octopus was designed by digital artist `parkjisun` and is available from the Noun Project [1]. No octopuses (or humans) were harmed in the making of this dissertation.

[1] `https://thenounproject.com/icon/octopus-469926/`

Appendix R

Author Biography

Photograph © Alexandra Meseke Photography, 2022

Benjamin S. Meyers received a B.S. in Software Engineering from the Rochester Institute of Technology (RIT) in 2018. He completed his Ph.D. in Computing and Information Sciences at RIT in 2023. His academic research focuses on human factors in software engineering and computer security, with special interest in linguistic characteristics of developers' conversations and software developers' human errors. Ben is currently a Research Computing Facilitator in the Research Computing Department at RIT, where he applies his software engineering, data science, research, and teaching skills to assist graduate students and faculty at RIT with their research.

Outside of his schooling, Ben is an avid reader of non-fiction (*e.g.* climate change, racial & social justice, linguistics, psychology, fungi, astrophysics, speculative biology, mythology, big history), fantasy (*e.g.* J.R.R. Tolkien, Ursula K. Le Guin), science fiction (*e.g.* Douglas Adams, Neal Stephenson, Andy Weir, Liu Cixin), and comic books (*e.g.* Superman, Harley Quinn, Spider-Man, Daredevil). Beyond reading, Ben enjoys building with Lego®, brewing beer, drinking tea, cooking, hiking through nature, and playing board games/tabletop role-playing games.

`mailto:bsmits@rit.edu`
`https://github.com/meyersbs`
`https://www.linkedin.com/in/benjaminsmeyers/`
`https://scholar.google.com/citations?user=kKlI_1AAAAAJ`