MOONSTONE: Support for Understanding and Writing Exception Handling Code

Florian Kistner,[†] Mary Beth Kery,[‡] Michael Puskas,[§] Steven Moore,[‡] and Brad A. Myers[‡]
florian.kistner@tum.de, mkery@cs.cmu.edu, mmpuskas@gmail.com, stevenjamesmoore@gmail.com, bam@cs.cmu.edu

[†]Department of Informatics, Technical University of Munich, Germany
[‡]Human-Computer Interaction Institute, Carnegie Mellon University, Pittsburgh, PA, USA
[§]School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ, USA

***Abstract*—MOONSTONE is a new plugin for Eclipse that supports developers in understanding exception flow and in writing exception handlers in Java. Understanding exception control flow is paramount for writing robust exception handlers, a task many developers struggle with. To help with this understanding, we present two new kinds of information: ghost comments, which are transient overlays that reveal potential sources of exceptions directly in code, and annotated highlights of skipped code and associated handlers. To help developers write better handlers, MOONSTONE additionally provides project-specific recommendations, detects common bad practices, such as empty or inadequate handlers, and provides automatic resolutions, introducing programmers to advanced Java exception handling features, such as `try-with-resources`. We present findings from two formative studies that informed the design of MOONSTONE. We then show with a user study that MOONSTONE improves users' understanding in certain areas and enables developers to amend exception handling code more quickly and correctly.**

I. INTRODUCTION

Exceptions are a common error-handling language mechanism seen by many as an important cornerstone of building robust systems [1]. When software encounters failures, appropriate exception handlers enable the program to recover gracefully or terminate safely. However, an abundance of evidence from prior studies has shown that poor exception handling practices are extremely prevalent [2]–[4].

Why is the exception mechanism so susceptible to misuse? People have blamed developers [5], [6] and poor usability of language mechanisms [7], [8]. For instance, developers do not consistently document exception conditions, leading to confusion and bugs when other developers interface with their code [9]. From the language perspective, Java is one of few that support *checked exceptions*, whose propagation is constrained by rules checked by the compiler and which must be declared in a method's signature. This feature has been claimed to lead to difficulties in maintainability [10]. However, exceptions in general are a language feature across many modern languages. Exceptions may be universally problematic in that they introduce additional control flow that is largely hidden and can make it difficult to assess the behavior of the software system [1]. While novices particularly struggle with exception handling mechanisms [11], analysis of open-source projects has shown that bad exception handling practices are common across all levels of developer expertise [12].

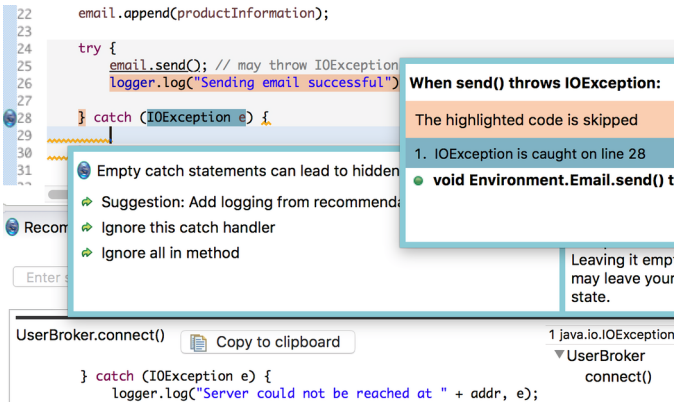

Fig. 1. Overview of MOONSTONE's features.

One powerful approach to address problems with both developer practices and language design has been tool support. Some prior research has designed tools to help with exceptions by allowing developers to visualize [13], [14] or set constraints on exceptions [15], [16] at a high level across their entire software project. These existing visualizations and constraint systems focus on helping fix design problems, but do not prevent developers from writing poor `try` or `catch` blocks in the first place. Many common problems with exceptions have been reported at the level of the code, such as catching exceptions too broadly or `catch` blocks that are simply empty [4]. Tool support is needed to help developers as they encounter exceptions in the code and make decisions about how to deal with them.

Given the well-documented struggles that novice and expert developers alike face with exceptions, tool support must more carefully consider the workflow of human developers. While prior research has mined software repositories and found patterns of exception handling use and misuse, here we take this data on *what* problems occur and interview developers to better understand *why* problems occur with exceptions. Our goal is to take a pragmatic and human-centered approach to these issues. We explore tool support to enable developers to solve exception handling tasks more quickly, write exception handling code that

This work was supported by a fellowship within the *FITweltweit* program of the German Academic Exchange Service (DAAD) and the National Science Foundation, Grant. No. CNS-1423054. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the sponsors.

978-1-5386-0443-4/17/$31.00

avoids poor practices and adheres to the project's exception handling policy, and to further the understanding of the Java exception handling mechanism and control flow.

We performed two initial qualitative studies with developers, first an interview study, followed by a survey which elicited responses from over 100 developers. Using our qualitative findings and issues highlighted by prior work, we designed and implemented MOONSTONE, which stands for "**M**aking **O**bject **O**riented **N**ovel **S**oftware **T**ools **O**ptimized for **N**oting **E**xceptions" (see Fig. 1). MOONSTONE is a plugin extension to the Eclipse[1] Java integrated development environment (IDE) that is designed to support the programmer in both understanding exception control flow and in writing and maintaining exception handling code.

MOONSTONE addresses two high-level problems with its six main features. The first problem is understanding control flow in the face of exceptions and how control flow relates to exception handlers. Although `try` and `catch` tokens visibly mark the code that handles an exception, there is no visible information for a programmer to tell what statements in the `try` block can potentially throw an exception, and which code would be skipped and which executed when an exception happens. We found this lack of visibility caused problems for programmers understanding what the exceptional situation was, and it also created ambiguity about how to appropriately handle the exception. To help with this problem, MOONSTONE provides two visualizations of intra-method control flow. To reveal potential sources of exceptions directly in the code, we provide transient overlays we call *ghost comments* that convey the exception information. Also, programmers can trigger annotated code highlighting of the consequently skipped lines of code and associated handling constructs by hovering over the ghost comment or the method call itself. This makes more accessible the information commonly needed when trying to understand exception handling code.

The second problem is that programmers tend to write poor quality exception handlers. To help users learn about and avoid bad exception handling practices, we implemented a set of static analyses for common problems and provide the user with detailed explanations of detected bad practices as well as automated fixes. We also added a lightweight recommendation engine that supports users when writing new exception handlers, and eases adhering to project-specific implicit exception handling policies.

We evaluated the effectiveness of these features with a user study comparing the performance of users using the standard version of Eclipse to users using Eclipse with the MOONSTONE plugin. Users in the experimental condition were able to complete the tasks on revising existing as well as writing new exception handling code quickly and accurately, whereas users in the control were both slower and scored lower.

The contributions described in this paper are:

- Two formative studies about developer behavior that showed that developers have a tendency to make use of blanket exception handlers and struggle to determine sources of exceptions in code;
- New visualizations and interaction techniques that support understanding of local control flow and authoring of higher quality exception handlers along with a user study confirming their usability.

[1]https://eclipse.org/ide/

II. RELATED WORK

A. Support for Understanding Exception Control Flow

Several prior tools have tried to help developers better understand exception handling. These tools typically focus on depicting control flow of exceptions across an entire program. For example, Chang et al.'s interactive tree menu allows navigating from the method throwing an exception to its `catch` sites [17]. Robillard and Murphy's static analysis tool, JEX, creates an abstract representation of the code that only shows sources of exceptions [13]. This still leaves it up to the user to determine the exception flow and transfer gained insights back to the actual source code. Shah et al. presented ENHANCE, a tool to provide visualizations about `throw-catch` pairs across a project in the user's IDE [14]. Both JEX and ENHANCE visualize exception control flow graphs [1] and allow developers to investigate overall exception flow and identify irregularities in exception handling. However, this will typically result in a large number of visualized flows that need to be considered and can therefore lead to overwhelming or confusing graphical representations. These prior tools also notably lacked any controlled studies with developers. In contrast, our approach does not rely on graph visualizations. Instead, we insert information about the local exception flow and about the affected code directly in the source code.

Prior work has found that organizations tend to not pay attention to exception handling, which creates a fertile ground for missing subtle details such as properly closing resources [3]. In an experiment on paper, Maxion found that simply by focusing developers' attention on error handling, they considered more details of the error handling control flow [18]. Motivated by these findings, our goal in MOONSTONE is to provide context-sensitive information directly in the developer's view of the source code. We focus on just-in-time support that will guide developers during code writing and maintenance tasks, rather than relying on external visualizations or tools a developer might consult later.

A few tools have approached exceptions as a specification and verification problem [15], [16]. These allow a developer to express exception handling policies through annotations, such as putting constraints on which exceptions can be thrown by a given class [16]. This prevents certain problems by disallowing overly general exceptions from being thrown at a method, class, or package's interface. Again, our focus in MOONSTONE is to provide local-level support to developers writing, reading, and maintaining exceptions rather than working at the software architecture level.

B. Support for Writing Exception Handling Code

Professional developers can use of a broad range of program analysis tools that aim to detect software defects, including some issues with exception handling code. While tools like

FindBugs [19] do detect some exception handling issues, they do not always provide fixes, and they produce explanations that assume the developer understands the problem. MOONSTONE provides background information supporting the user in gaining a better understanding of the problems and for making an informed choice about how to proceed. We also provide context-sensitive automatic resolutions to introduce programmers to idiomatic solutions they may not be aware of.

Recently, recommendation systems to suggest adequate exception handlers from a large corpus using machine learning techniques have been demonstrated [20]. However, due to the computational complexity that the training requires, they are currently limited to common libraries for which the system can be trained in advance. Therefore, they cannot practically be employed for project-specific recommendations. Other systems suggest examples that are structurally similar to the referenced code based on the assumption that structural similarity and relevance are correlated [21]. A limitation of this work is that the authors pulled recommendations from only one source: projects under the Eclipse foundation, which are likely to have their own implicit exception handling policy. The authors do not demonstrate that these match well to the policies of any other software project. Furthermore, they detect and purposely exclude exception handlers that contain "bad practices" from the recommendations. However, our studies with developers suggest that, at least in some cases, these exception handlers, for example ones that contain only logging, are what is intended. Logging is very often used to provide key information for unrecoverable exceptions. In MOONSTONE, we leverage the statistical local similarity of exception handlers [4] instead of the structure of the surrounding code as basis for our light-weight recommendation system, which allows us to provide support for local, implicit exception policies.

III. BACKGROUND

In programming, an exception is a term for an "exceptional event." We make the common distinction between contingencies, which are routine failures the programmer should anticipate and be prepared for, versus faults, which cover all other failures that the programmer cannot anticipate or are highly improbable [22]. Examples of contingencies are a missing file or an unreachable server, while faults are typically programmer errors or unexpected failures of a system outside of the control of the program itself.

Java uses the conventional `try-catch-finally` mechanism. When an exception is thrown, the runtime interrupts the normal program control flow, and unwinds the call stack until a `catch` block is found that can *handle* (deal with) that exception, executing all `finally` blocks passed through during the search.

As shown in Fig. 2, Java includes different exception base classes that determine whether an exception is *checked* and thereby subject to additional constraints in the type system. Thrown checked exceptions must be declared in the signature of a method or be handled by a `catch` handler. While there are some people that advocate avoiding these stronger requirements entirely [8], contingencies are conventionally expected

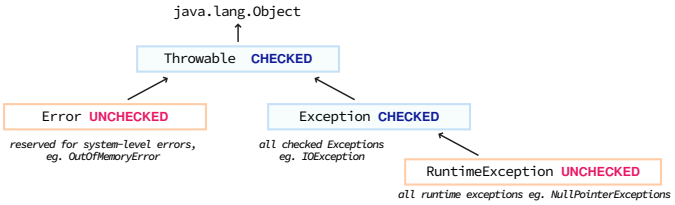

Fig. 2. Taxonomy of exception base classes in Java.

to be modeled as exceptions derived from the `Exception` class, so that the type system ensures that callers provide a corresponding handler. In contrast, the `Error` class is reserved for severe system failures, and other faults are expressed as `RuntimeException`. The latter commonly occur as a result of unexpected bugs such as a `NullPointerException`.

A consequence of exceptions is that, even if programmers do not think an exception is likely to be thrown, they must ensure resources are cleaned up in case an exception does happen [23]. Otherwise, stale resources can lead to resource starvation, since there are no guarantees that Java's garbage collector will collect involved objects and release underlying resources. Starting from version 7, Java offers a language construct to address this common scenario. The `try-with-resources` statement introduces scoped exception-safe resource management for objects implementing the `AutoCloseable` interface. Java deterministically invokes the `close()` method of objects managed in this manner when their declaring scope is exited, similar to the "Resource Acquisition Is Initialization" idiom seen in languages like C++, Ada, or Rust. This absolves programmers of writing explicit exception handling code in the form of correct `finally` handlers, while also covering difficult edge cases such as exceptions thrown during cleanup.

IV. FORMATIVE INVESTIGATION

We conducted two qualitative studies with professional developers to better understand *why* bad practices occur in their code base, and how developers approached exceptions. First, we interviewed 5 software developers (2 female, 3 male). Prior work suggests that developers in their first few years have more troubles using exceptions than seasoned developers [24]. In order to capture a greater variety of difficulties developers face, we therefore chose to examine three "novice" developers with only 1 to 2 years of professional development experience as well as two senior developers with at least 8 years of professional experience. All had at least 5 years of general programming experience. While we do not claim that this small sample is representative of all programmers, these interviews were revealing and helped to guide our design of a broader survey.

A. Interview

Interviews lasted up to an hour. Participants were not compensated for their time. Following basic demographic questions, the interviewer asked the participant to freely talk about their experience with exception handling. Next, participants were asked structured questions about how they learned about exception handling. Then, the experimenter gave each participant three examples of exception handling code on paper. The first

had a blank `catch` block, which participants were asked to fill in. The second and third were full examples that included bad exception practices, such as throwing `Throwable`, which is generally considered too general of a class. Participants were asked to review these two, and were then asked to comment on any bad practices they noticed (if they did not mention them on their own). In all three tasks, the participants were asked to think-aloud to reveal their thought processes.

The results show that, above all, the primary concern of all participants was logging. Many participants talked about exceptions occurring as primary signs of a bug. Participants discussed frustrations where another developer had left an insufficient log message or had thrown too general of an exception, like `Exception`, leaving no clues about the nature of the problem. Participants also discussed problems in the debugging process in trying to find the place in the code base that originally threw the exception. Often this required searching through each line in a `try` block to figure out which one had thrown the error:

> *"Usually, I like to try to have try blocks be as small as possible, just to limit the number of catches you have to have, and limit the amount of searching you have to do before you can figure out which line actually caused the exception to be caught."* – P1

Contrary to findings by Shah et al. [24], the three novice developers in our interviews did not express an "ignore for now" approach to exception handling. Rather, their attitude towards exception handling depended very much on the culture of their development team and company. Some participants had experienced intensive code reviews of their code, even on their log messages in exception handlers. One participant admitted to frequently catching and throwing `Exception`, even though they knew it was a bad practice, because the rest of the team's codebase followed these kinds of bad practices anyway.

B. Survey

We next designed an online survey based on questions from the interviews. The survey was sent out via email lists and social media accounts of the investigators. Participants were required to have some professional experience as a software developer, and were offered optional entry into a raffle for a $25 gift card for their participation.

A total of 101 developers responded to the survey, with an average of 13.9 years of professional development experience (SD = 12.6). Overall, developers were confident about exception handling, with 92% of participants responding that they felt they knew exception handling "Reasonably well" (51%) or "Very well" (41%). Survey participants had learned exception handling from a highly diverse set of sources. This corresponds with previous findings [22] that exception handling is not a topic well covered by formal courses, so people need to learn it elsewhere.

Despite confidence with exceptions, survey participants reported following poor practices at times. 62% of participants reported catching general exceptions like `Exception` or `Throwable` at least "Sometimes." Participants did not typically leave exception handlers empty, with 45% reporting that they "Never" do this, 31% reporting "Rarely," 17% "Sometimes," 4% "Often," and 3% "Very Often." If a participant answered more than "Never," we asked in a free response question for reasons or circumstances when they do this. Responses fell into two general themes: participants left empty `catch` blocks when they were writing temporary or prototype code. Participants also said they left `catch` blocks empty when they simply wanted to silence any exceptions that occurred at that program point.

To understand exception handling while reading code, 43% of participants reported that they found it at least "Somewhat Difficult" to determine which statement in a `try` block could potentially throw an exception. 76% reported at least "Sometimes" looking up the documentation of individual method calls to figure out which exceptions they could throw. 71% reported at least "Sometimes" trying to purposely minimize the size of a `try` block for this reason.

V. Tool Design

Moonstone is built using the plugin interface of Eclipse, which is a popular development environment for Java [25]. After designing and building a prototype of the tool based on our qualitative findings and prior work, we additionally tested the prototype with five developers to discover usability flaws and iterated based on that feedback. Then, we performed an extensive user study of the final design (see section VI).

A. Understanding control flow

Exceptions can reroute normal control flow in a way that developers find difficult to understand. While tokens like `try`, `catch`, `finally`, and `throw` provide important information, the control flow in the end depends on which methods throw which exceptions and what type hierarchy these exceptions have. As found in our survey and interviews, programmers are having difficulties with gathering this information.

```
28      try {
29          server.openConnection(addr); // may throw IOException
30          logger.log("connection opened"); // log for troubleshooting
31
```

Fig. 3. Ghost comment (in gray) on call to `server.openConnection`.

First, to help programmers identify potential exception sources, we augment the source code with the exception signatures of methods. We overlay these next to method invocations in form of what we call *ghost comments*. As seen in Fig. 3, we use the same font and size as the rest of the code to display a syntactically valid Java line comment at the end of the line of the call. Previous work found similar comment beacons to be supportive of program understanding and easy to process for experienced programmers [26]. To reduce visual clutter, we only display these ghost comments context-sensitively for method calls inside a `try` block, while the caret resides inside the relevant `try`, `catch` or `finally` blocks. In addition, the user can choose to persist an annotation, turning the ghost comment into an actual comment in the source file. After feedback to our initial prototype we also implemented ghost comments for the implicit calls to the `close()` method of a resource managed with a `try-with-resources` statement.

In addition, as shown in Fig. 4, we provide precise information about the resulting exception flow, which appears when

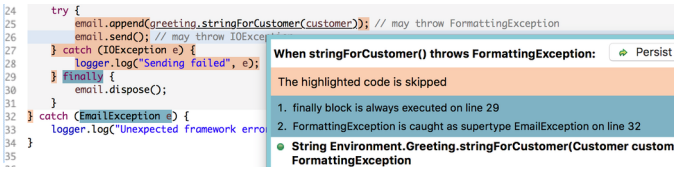

Fig. 4. Visualization of skipped code and associated handlers after hovering over `greeting.stringForCustomer` call.

users hover over a ghost comment or over a call that might throw an exception. This includes apricot colored code highlights of lines skipped due to an exception as well as blue code highlights of associated local exception flow constructs. MOONSTONE is capable of showing multiple exception flows originating from a single method call, for instance if a method declares multiple exception types. In this case users can select the flow they want to visualize. Our approach uses static analyses to determine all possible control flows due to checked exceptions. This includes exceptions on method calls whose return values are used as arguments as seen in Fig. 4, exceptions on implicit calls to `close()`, as well as exceptions in `finally` handlers. If the responsible exception handler cannot be statically determined, we narrow the resulting flow to the different paths possible at runtime.

Also shown in Fig. 4 is our custom hover popup window. It extends the standard JAVADOC information Eclipse provides with a color-coded legend and a step-by-step listing of the control flow through associated handling constructs corresponding to the blue highlights, such as `finally` handlers and `catch` handlers related to the user-selected throw.

B. Educating about bad exception handling practices

To promote a better understanding of exception handling problems, we extend upon Eclipse error and warning features. We introduce a popup window (see Fig. 5) that provides an in-depth explanation of the problems encountered and context-sensitive automated quick fixes to help users make use of appropriate Java features and improve their exception handling code.

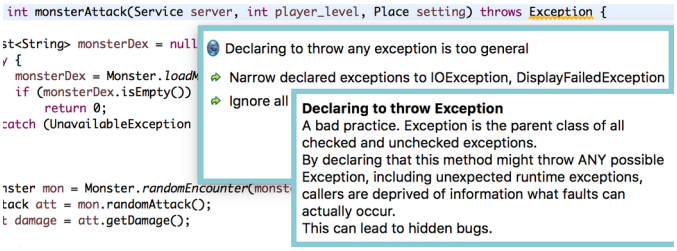

Fig. 5. Explanation of the bad practice of declaring base class `Exception` on method with suggestions for context-sensitive resolution shown side-by-side.

The implementation of this functionality uses a set of seven static analysis passes. We flag overly generic `catch` handlers that assume responsibility for all exceptions deriving from one of the exception base classes `Throwable`, `Error`, `RuntimeException`, or `Exception`, if the type checker only requires a subset to be handled. This addresses the common theme found in our investigation of blanket `catch` handlers without differentiation of the failure scenario. While dealing with all checked exceptions in a uniform way can be reasonable for some code, an `Exception` or `Throwable` `catch` handler also responds to all `RuntimeExceptions` (see Java's exception hierarchy in Fig. 2), including subtle failures like arithmetic overflows or programmer errors which often require special care to recover from appropriately. An exception handler written to handle contingencies might thereby also inappropriately handle these failures. Analogously, we report overly generic exception declarations on methods, when a narrower exception specification can be provided to the caller. First, an overly generic exception specification leaves callers no choice but to implement a blanket `catch` handler additionally contributing to the problem of inadvertently handled unchecked exceptions. Second, as discussed in the previous section, accurate information about possible exceptions plays an important role in writing exception handling code. Generic exception specifications do not provide this. Further analyses detect throws of base class exceptions, empty `catch` handlers, as well as the common mistake of failing to provide the underlying exception as the cause when throwing a new exception, resulting in loss of the stack trace up to that point.

Lastly, we flag failures to properly clean up resources that implement the `AutoCloseable` interface. While this problem has been addressed in tooling before, many are unfamiliar with the semantics of newer Java features such as the `try-with-resources` statement and are therefore reluctant to make use of it or are prone to misuse it. Our detailed explanation in combination with the ghost comment informing the user about the implicit `close()` resource cleanup call should ease the transition to using this language feature and thereby to writing more resilient resource cleanup code.

For our quick fix feature, we employ transformations of the program's abstract syntax tree (AST) to reliably perform the change the user selected. This ensures broad compatibility with real world source code. After feedback about our initial prototype, we also implemented functionality to help users with following how their code was affected. We use information available from our AST transform to steer attention to newly introduced constructs by changing the text selection to the corresponding code in the source code editor.

C. Supporting maintenance

Prior research indicates that much of exception handling code is rarely tested [3]. This puts additional strain on maintaining correct exception handling in the code. Using explicitly specified exception policies to guide the recommendations has been explored [15]. However, establishing consistent exception policies can be challenging and may require refactoring part of the software system, which renders this approach impractical for smaller projects or less experienced developers. Based on the previous work that reported that exception handlers have a tendency to be similar to other handlers in their vicinity [4], we implemented functionality to support adherence to local, implicit exception handling policies.

We provide the programmer with examples of other exception handlers from the current project that deal with a certain exception type as well as examples of typical `finally` handlers. This

allows the user to get an overview of existing exception handling code and helps to discover required recovery procedures as well as project-specific conventions, such as the employed logging scheme, or the granularity of errors reported. Available matching examples are shown in an additional pane in the user's IDE (see bottom of Fig. 1), which automatically updates as the user moves the text cursor in the source file. To lower the burden of implementing appropriate handlers rather than leaving the `catch` block empty, we provide functionality to copy an existing handler into the system's clipboard to make it easy to paste it as the basis for a new implementation. We gather the information needed for these recommendations by indexing all exception handling constructs in the current project, when the user first opens it in the IDE. When changes to the code are detected, the index is updated accordingly. We use a code transformation that abstracts non-structural details, such as variable names and string constants, to group similar handlers together in order to eliminate recommendations without significant differences in their control flow or functionality.

Our quick fixes also play an important role in maintaining the software system by helping developers avoid introducing new bad practices. To that end, our recommendation system is tightly integrated with our poor practice detection. For instance, we enable users to jump directly to a matching recommendation to resolve an empty `catch` handler. We categorize the indexed handlers to allow users to quickly jump to examples that are appropriate for the current situation, such as handlers involving logging code or those that do not.

VI. Evaluation

To evaluate the usability and effectiveness of Moonstone, we conducted a laboratory study of experienced Java programmers working on a range of coding tasks. We used the *neon.1* version of the Eclipse Java IDE. The control group used the default configuration, whereas the experimental group used the same version of Eclipse extended with our Moonstone plugin. The effects of our plugin were measured using a between-subjects study design, where participants were equally distributed among the two conditions.

A. Participants

Most of the participants were students (85%) recruited from the University of Pittsburgh and Carnegie Mellon University based on their software development experience. We accepted participants into the study that either self-identified as having more than three years of Java experience or had experience working with large Java software projects outside of a university. Additionally, we required experience in using an integrated development environment, not limited to Eclipse. Each participant was paid $20.

We chose to exclude one participant of the control group whose knowledge of the Java exception handling mechanism was not in line with the self-identified experience and who seemed to not meet the requirements. Thus, our study consisted of 14 participants (8 male and 6 female), eight of whom were graduate students. All but one participant had experience working in software development with seniorities ranging from $\frac{1}{2}$ to 9 years. Overall, they reported, on average, 3.2 years of professional experience (SD = 2.4) and 4.8 years of experience in Java (SD = 2.4).

B. Tasks

To increase the external validity of the tasks, we collected a series of bug reports, commits of bad practices, and commits fixing bad practices in a number of open source projects. Based on common problems we encountered, we created a set of 13 tasks which used simplified versions of those situations or similar problems, covering the different aspects our plugin addresses:

- 2 tasks about *identifying* problems and bad practices in the given piece of code;
- 5 tasks about *fixing* incorrect exception handling code and poor practices, covering failures to correctly cleanup resources on all execution paths, empty `catch` handlers, usage of overly generic exception types, and failures to preserve the original exception as the cause when throwing a new exception in `catch` handlers;
- 4 tasks about *understanding* the control flow in the face of exceptions, including identifying exception sources as well as inconsistencies between code intent and given exception handlers and comprehension of `finally` semantics;
- 2 tasks about *implementing* new `catch` handlers that are in line with how other exception handlers in the same project handle problems.

All participants did all the tasks in the same order. Dependent measures were time to completion and success. To provide a more nuanced evaluation of success, we devised a grading rubric that assigns 2 to 7 points to each task based on its complexity and clear-cut features, such as the absence of compiler errors or achieved program semantics. We imposed time limits per task depending on the section they were in to keep participants from getting caught up in one of the tasks. For the identifying and understanding tasks, the time limit for each task was five minutes, for the fixing tasks six minutes, and for the implementation tasks eight minutes. If a participant was still working on a task when the time limit was reached, we evaluated their progress up until that point, but excluded their time from further analysis.

C. Procedure

The study had a total length of 90 minutes. At the beginning, we provided participants with short introduction of similar length for the version of Eclipse they were assigned to use. Participants were then encouraged to try out the explained features themselves and were given appropriate help when they had problems triggering features of Moonstone or Eclipse, such as hovering over elements with the touch pad of the provided laptop computer or specifics of the macOS operating system used for this study. Information covered in the introduction included Eclipse's features to navigate around code, such as "Open Declaration," as well as the JavaDoc text hovers to discover exceptions declared by a method (for the control condition), and the ghost comments, places where our extended text hover can be triggered, as well as an explanation how to apply quick

fixes (for the experimental condition). The participants then proceeded to work on the first two sections of our user study.

Before we moved to the understanding and implementation sections, we briefed participants on another set of features to prepare them for the remaining tasks. In the control condition, participants were shown Eclipse's project explorer which provides an overview of the project structure and semantic outlines of declared Java elements as well as the search functionality to either limit searches to certain types of identifiers or initiate a plain text search. Participants in the experimental condition were given an explanation of the information MOONSTONE's text hovers provide as well as the recommendation feature. After completing the tasks, the participants were asked to fill out a survey about the version of Eclipse they used.

D. Results

There were 7 participants in each condition who worked on all tasks. As Fig. 6 shows, participants using MOONSTONE finished on average faster in each group of tasks, and overall. In total, 16.5% of tasks in the control condition as well as 2.2% in the experimental condition were not finished within the time limit and were excluded from Fig. 6. The values on each bar in Fig. 6–8 denote the number of data points taken into account.

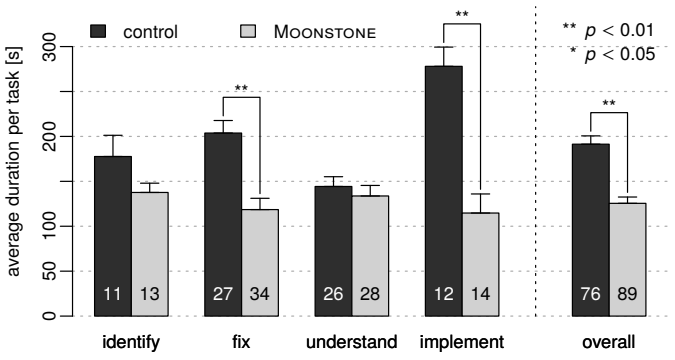

Fig. 6. Average duration per finished task in section in tasks analyzed.

A Mann–Whitney U test confirms that the increase in speed is significant for the tasks that involve fixing problems ($U = 182.5$, $p < 0.01$) or implementing new handlers ($U = 13.0$, $p < 0.01$). The difference in the identify tasks was not significant, but the high difference overall (−22%) suggests that statistical significance might have been achieved with more participants. There did not seem to be any difference in speed for understand tasks.

As Fig. 7 shows, participants in the experimental treatment achieved higher scores on fixing problems ($U = 28.0$, $p < 0.01$) and implementing new exception handlers ($U = 28.5$, $p < 0.05$). The differences in identified problems and scores for understanding control flow show similar tendencies, but were not significant, what we similarly attribute to limited number of people in this user study.

For further analysis, we also assigned the individual issues participants needed to address into the following categories: *syntax and logic* errors, making use of *specific* exception *types* when catching or throwing exceptions instead of one of the exception base classes, correctly cleaning up *resources*, and

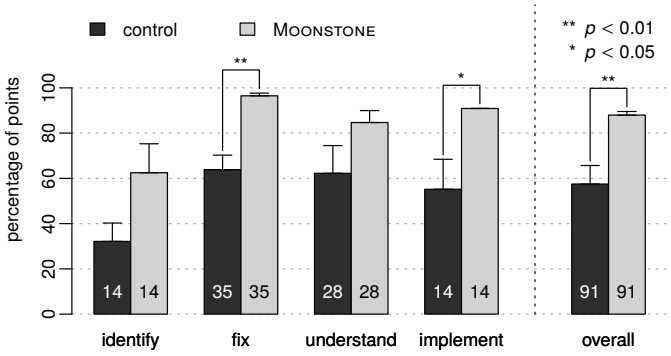

Fig. 7. Average percentage of points per section in tasks analyzed.

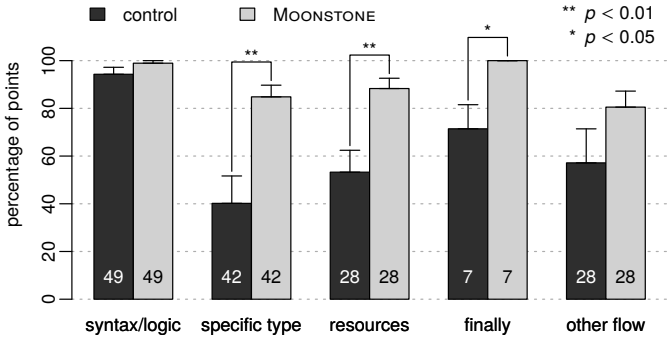

Fig. 8. Average percentage of points per category in tasks analyzed.

knowledge of control flow associated with *finally* constructs as well as *other* non-`finally` related exception control *flow*, see Fig. 8. Based on this classification, MOONSTONE was most helpful for tasks involving handling of resources ($U = 44.0$, $p < 0.01$) and understanding how the `finally` handler interacts with exception control flow ($U = 38.5$, $p < 0.05$). Similarly, it also encouraged developers to use significantly more specific exception types in their code ($U = 44.0$, $p < 0.01$).

In the survey at the end of the study, MOONSTONE users emphasized the value of extra information provided by the plugin, such as the ghost comments and code highlights, as well as the recommendation feature and the support MOONSTONE provides to ensure consistent exception handling within a project. When asked, all participants in the experimental condition showed interest in using such a tool. Users gave MOONSTONE an average rating of 6.1 for ease of learning and 6.3 for overall likability on a 7-point Likert scale compared to an ease of learning rating of 5.1 and overall likability of 5.6 in the control condition.

VII. DISCUSSION

Our results show that users found MOONSTONE easy to use and were satisfied with the quality of information provided. MOONSTONE helped programmers to complete exception handling coding tasks significantly faster, and to fix poor exception handling practices and problems and to correctly implement new project-specific exception handlers to a significantly higher degree. Further, developers exhibited an improved understanding of exception control flow in the face of `finally` handlers.

A. Qualitative Observations

Despite the instructions about MOONSTONE, some participants showed hesitation to use the tool at first. Several were reluctant to use the quick fixes, but still consulted the plugin for information about the type of problem diagnosed. One participant cited his distaste for auto-generated code as the motivation. Nevertheless, overall, participants performed considerably better using MOONSTONE for the fixing tasks, suggesting the information in the quick fixes were a benefit even to users who did not apply them.

In the understand section, we found that a few participants from both conditions experienced difficulties triggering Eclipse's text hover functionality which MOONSTONE builds upon. Affected users would hover for a short time over the target before initiating a click on the target, which caused the hover to be dismissed before the information could be reviewed. Eclipse allows for configuring the time until the hovers are triggered suggesting a shorter interval might improve usability. Additionally, typing on the keyboard did not dismiss MOONSTONE's hovers if the mouse was still hovering over it, which was unexpected to some users. This seems to have affected the results in the understanding section for the experimental condition.

Participants in the control condition faired especially poorly in the implement tasks. Even though Eclipse's project explorer and search functionality were highlighted, participants struggled to form a plan about where to find other exception handlers in the project and thus spent a lot of time reviewing the source files and declarations of the elements referenced.

B. Limitations

The user study was performed with a small number of participants most of whom were university students. Even though some of them reported considerable experience in professional software development, they may not be representative of other Java developers. Further, we did not verify their level of proficiency with an assessment. A larger user study might therefore be appropriate to explore applicability to a broader audience. Similarly, while the tasks are based on selected common exception handling problems, the tasks were synthetically designed for this laboratory study. Therefore, they may not be representative of more complex real-world code.

MOONSTONE's highlighting of control flow provides robust, precise information, but relies on static analysis to determine possible flows. If programmers use exception handlers to differentiate between subtypes of a declared checked exception, the actual exception flow may not be determinable until runtime. MOONSTONE displays all possible concrete exception types resulting in different flows in that case, but we feel that an appropriate mechanism to rank potential flows according to their likelihood of occurrence is needed to be an effective tool for programmers when this happens, since it can display exception flows that are impossible with the current implementation.

Furthermore, since MOONSTONE's recommendations are sourced from the current project's files, they are of limited utility when starting a new project or when existing handlers are of low quality. However, MOONSTONE could be extended to allow developers to choose a previous project of a similar domain or a repository of best practice handlers to bootstrap or augment the recommendations based on the current project.

VIII. FUTURE WORK

MOONSTONE could be further extended in numerous ways, such as support for other poor practices, more complex analyses of exception propagation, as well as broadening the analyses of too generic types. This might provide more comprehensive support for software developers resulting in further improvements in the quality of exception handling code.

Additionally, a global analysis, such as the one employed by JEX [13], could be used to expand its functionality to unchecked exceptions, whose propagation is not recorded in the type system. Most of the presented ideas can therefore be adapted for other IDEs and imperative languages, even if the target environment does not have checked exceptions.

To measure and improve upon the benefit to novice software developers, a longitudinal study of MOONSTONE in a university class setting over the course of a semester would be a good follow-up for future work.

Although powerful, tool support, as explored in this paper, will always be constrained by the existing semantics of the language. Another approach can be changes to the programming language itself. While participants exhibited a bias for fine-grained exception handling according to our findings, others have proposed separating exception handling responsibilities further from regular software developers [6]. To that end, our current work is exploring a combination of language changes and human-centric tool support to establish fault barriers [22], which are exception handlers that implement worst-case recovery, and may be able to provide the exception handling guarantees that are required to have robust software systems.

IX. CONCLUSION

In this paper, we introduced MOONSTONE, which supports programmers in understanding and writing exception handling code. Our formative investigation revealed that developers spend a fair amount of time searching for potential sources of exceptions, up to the point where they consciously try to minimize `try` blocks to limit the search space, and many developers make use of blanket `catch` handlers and/or leave `catch` handlers empty. We designed MOONSTONE based on the data we collected and iterated on our prototype based on user feedback from software developers, resulting in a system that helped users understand and better write exception handling code.

Dealing with exception handling code continues to be a time-consuming and mentally challenging task. MOONSTONE points the way for how developer productivity can be improved with support for basic problems programmers face, thereby freeing mental capacity for higher level tasks.

> *"It allows me to understand and locate problems faster, optimizing my programming performance"*
>
> – participant in the experimental condition

REFERENCES

[1] M. P. Robillard and G. C. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 2, pp. 191–221, 2003.

[2] B. Cabral and P. Marques, "Exception handling: A field study in java and .net," in *European Conference on Object-Oriented Programming 2007 (ECOOP '07)*. Berlin, Heidelberg: Springer, 2007, pp. 151–175.

[3] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in java programs," *Journal of Systems and Software*, vol. 106, pp. 82–101, Aug. 2015.

[4] M. B. Kery, C. Le Goues, and B. A. Myers, "Examining programmer practices for locally handling exceptions," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. New York, NY, USA: ACM, 2016, pp. 484–487.

[5] H. Shah, C. Gorg, and M. J. Harrold, "Understanding exception handling: Viewpoints of novices and experts," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 150–161, Mar. 2010.

[6] H. Shah and M. J. Harrold, "Exception handling negligence due to intra-individual goal conflicts," in *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering (CHASE '09)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 80–83.

[7] A. P. Black, "Exception handling: The case against," Ph.D. dissertation, University of Oxford, 1982.

[8] B. Eckel. (2001, Oct.) Does java need checked exceptions? [Online]. Available: http://www.mindview.net/Etc/Discussions/CheckedExceptions

[9] M. Kechagia and D. Spinellis, "Undocumented and unchecked: Exceptions that spell trouble," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*. New York, NY, USA: ACM, 2014, pp. 312–315.

[10] D. Reimer and H. Srinivasan, "Analyzing exception usage in large java applications," in *Proceedings of ECOOP'2003 Workshop on Exception Handling in Object-Oriented Systems*, 2003, pp. 10–18.

[11] R. Rashkovits and I. Lavy, "Students' misconceptions of java exceptions," in *Proceedings of the 7th Mediterranean Conference on Information Systems (MCIS '12)*. Berlin, Heidelberg: Springer, 2012, pp. 1–21.

[12] M. Asaduzzaman, M. Ahasanuzzaman, C. K. Roy, and K. A. Schneider, "How developers use exception handling in java?" in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. New York, NY, USA: ACM, 2016, pp. 516–519.

[13] M. P. Robillard and G. C. Murphy, "Analyzing exception flow in java programs," *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 322–337, Nov. 1999.

[14] H. Shah, C. Görg, and M. J. Harrold, "Visualization of exception handling constructs to support program understanding," in *Proceedings of the 4th ACM Symposium on Software Visualization (SoftVis '08)*. New York, NY, USA: ACM, 2008, pp. 19–28.

[15] E. A. Barbosa, "Mastering global exceptions with policy-aware recommendations," in *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, vol. 2. Piscataway, NJ, USA: IEEE Press, 2015, pp. 778–780.

[16] D. Malayeri and J. Aldrich, "Practical exception specifications," in *Advanced Topics in Exception Handling Techniques*, ser. Lecture Notes in Computer Science, C. Dony, J. L. Knudsen, A. Romanovsky, and A. Tripathi, Eds. Berlin, Heidelberg: Springer, 2006, pp. 200–220.

[17] B.-M. Chang, J.-W. Jo, and S. H. Her, "Visualization of exception propagation for java using static analysis," in *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*, 2002, pp. 173–182.

[18] R. A. Maxion and R. T. Olszewski, "Improving software robustness with dependability cases," in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*. IEEE, Jun. 1998, pp. 346–355.

[19] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*. New York, NY, USA: ACM, 2007, pp. 1–8.

[20] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. New York, NY, USA: ACM, 2014, pp. 419–428.

[21] E. A. Barbosa, A. Garcia, and M. Mezini, "Heuristic strategies for recommendation of exception handling code," in *26th Brazilian Symposium on Software Engineering*, Sep. 2012, pp. 171–180.

[22] B. Ruzek. (2007, Oct.) Effective java exceptions. [Online]. Available: https://oracle.com/technetwork/java/effective-exceptions-092345.html

[23] J. Schwille, "Use and abuse of exceptions — 12 guidelines for proper exception handling," in *12th Ada-Europe International Conference (Ada–Europe '93)*, M. Gauthier, Ed. Berlin, Heidelberg: Springer, 1993, pp. 142–152.

[24] H. Shah, C. Görg, and M. J. Harrold, "Why do developers neglect exception handling?" in *Proceedings of the 4th International Workshop on Exception Handling (WEH '08)*. New York, NY, USA: ACM, 2008, pp. 62–68.

[25] D. Geer, "Eclipse becomes the dominant java ide," *Computer*, vol. 38, no. 7, pp. 16–18, Jul. 2005.

[26] R. L. Shackelford and A. N. Badre, "Why can't smart students solve simple programming problems?" *International Journal of Man-Machine Studies*, vol. 38, no. 6, pp. 985–997, 1993.