What Would Other Programmers Do? Suggesting Solutions to Error Messages

Björn Hartmann[1], Daniel MacDougall[2], Joel Brandt[2], Scott R. Klemmer[2]

1–Computer Science Division
University of California, Berkeley, CA 94720
bjoern@cs.berkeley.edu

2–Stanford University HCI Group
Computer Science Dept, Stanford, CA 94305
{dmac,jbrandt,srk}@cs.stanford.edu

ABSTRACT

Interpreting compiler errors and exception messages is challenging for novice programmers. Presenting examples of how other programmers have corrected similar errors may help novices understand and correct such errors. This paper introduces *HelpMeOut*, a social recommender system that aids the debugging of error messages by suggesting solutions that peers have applied in the past. *HelpMeOut* comprises IDE instrumentation to collect examples of code changes that fix errors; a central database that stores fix reports from many users; and a suggestion interface that, given an error, queries the database for a list of relevant fixes and presents these to the programmer. We report on implementations of this architecture for two programming languages. An evaluation with novice programmers found that the technique can suggest useful fixes for 47% of errors after 39 person-hours of programming in an instrumented environment.

Author Keywords: debugging, recommender systems

ACM Classification: H.5.2 [Information Interfaces and Presentation]: User Interfaces – Training, Help, and Documentation. D.2.5 [Software Engineering]: Testing and Debugging – Debugging Aids.

General terms: Design, Human Factors

INTRODUCTION

Programmers—especially amateurs—often create software by opportunistically modifying found examples [5], and they regularly use online forums and blogs to seek help. However, most development tools remain largely unaware of this social life of code and lack explicit support for it.

Using the web as a medium for sharing code and seeking code-specific help clearly has value; it also has important limitations as a platform. Standard search engines index string literals rather than code semantics, making it hard to specify queries for code. Specialized code search engines incorporate language semantics, but they mainly index repositories of *working* code bases, making them less helpful for debugging tasks. Many programmers thus post questions to online forums where answers may have high latency or may not be answered at all. We believe that there is significant latent value in integrating communal information exchange around debugging directly into authoring tools, where richer ways for collecting, presenting, and interacting with code are available.

As a step into the direction of integrating collective information into programming tools, this paper proposes *HelpMeOut*, a recommender system that aids novices with the debugging of compiler error messages and runtime exceptions by suggesting successful solutions to similar errors that other programmers have encountered.

Novice programmers have difficulty interpreting compiler errors [26]. We hypothesize that *presenting relevant solution examples makes it easier for novices to interpret and correct error messages.* Programming by example modification has been noted to be significantly easier to end-users than creation from scratch [27]; it has been documented in laboratory studies [6] and class observations [34] of student programmers. Examples present a concrete solution rather than an abstract problem statement. People are adept at solving problems by analogy [11] — we hypothesize that showing examples of related fixes enables such analogical problem solving.

The HelpMeOut system collects and suggests error corrections by augmenting existing programming development environments (IDEs). HelpMeOut comprises four components (see Figure 1):

1) Instrumentation that tracks code evolution over time and collects modifications that take source code from an error state to an error-free state ("fixes").
2) An online database for storing fixes which can be queried for most relevant examples, given an error message and code context.
3) A suggestion interface inside an IDE that presents a list of possible fixes for and error to the user, and aids with integration of a fix into her code.
4) A web interface to elicit and collect plain text explanations of collected fixes by experts.

The main contribution of this paper is a *new strategy of collecting and presenting crowdsourced suggestions for programming errors inside an IDE*. The paper contributes a general architecture for such a system, two implementa-

CHI 2010, April 10–15, 2010, Atlanta, Georgia, USA.

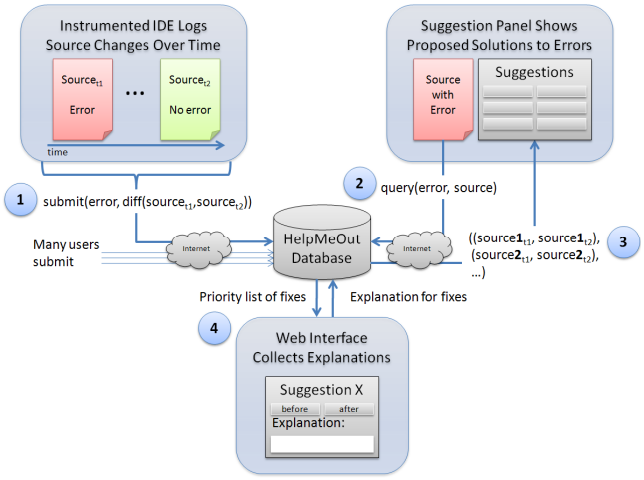

Figure 1. HelpMeOut offers asynchronous collaboration to suggest corrections to programming errors. **1**: IDE instrumentation collects bug fixes and sends them to a remote database. **2**: Other programmers query the database when they encounter errors. **3**: Suggested fixes are shown inside their IDE. **4**: Explanations for fixes are collected in a web interface.

tions, an initial evaluation and a discussion of the potential benefits and limitations vis-à-vis other approaches.

The fundamental technical insight enabling HelpMeOut is to *use both error messages and source code context in the capture and search for relevant fixes*. Instead of searching for source code using plain text, the code is tokenized using a custom lexical analyzer, which enables searching for common code structure across different projects.

HelpMeOut is influenced by past work on mining source code repositories retrospectively for bug finding [19,24]. Such work has generally focused on expert programmers and completely automatic bug finding and fixing methods. HelpMeOut also extends research on authoring environment instrumentation, which has been used to derive usage patterns [33] and to suggest commands [23,25].

The remainder of this paper is organized as follows: We first present a scenario that demonstrates the benefit of HelpMeOut; we then discuss architecture and implementation of its three principal components; discuss evaluation strategies, privacy implications and inherent limitations; and conclude with a review of related work.

SCENARIO

Jim, a design student in art school, works on code for an animation based on mouse input. In his code, he incorrectly initializes a variable array:

```
float[] x = new float[];
```

When trying to compile his code he receives the error message "*Variable must provide either dimension expressions or an array initializer*." Not sure what either of the two options mean, he consults the HelpMeOut suggestion panel (Figure 2). He sees that he can either add a size to the right-hand side of his variable intialization, or provide

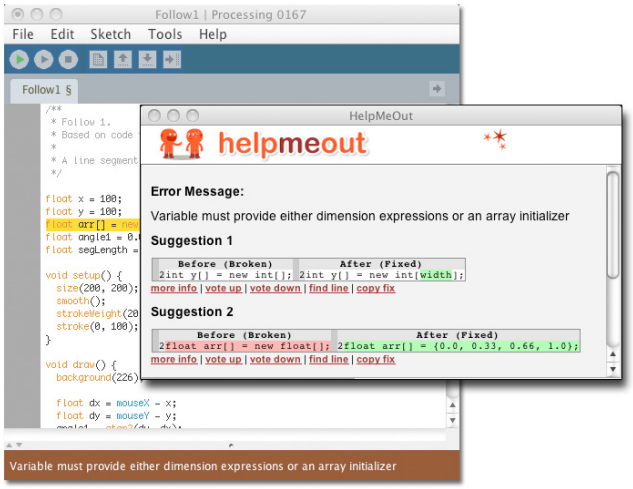

Figure 2. The HelpMeOut Suggestion Panel shows possible corrections for a reported compiler error.

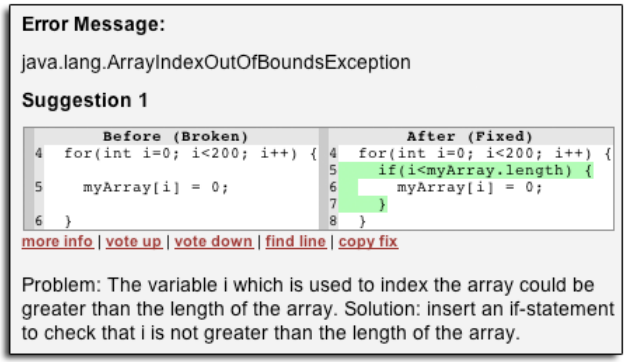

Figure 3. A suggestion for a runtime error which includes an explanation of the fix.

explicit values. He clicks on the "copy fix" button next to the first suggested fix, which modifies his original source line to add an array size, leaving his variable name and the rest of his code intact. He then changes the array size to fit his requirements.

His program now compiles, but at runtime an *ArrayOutOfBounds* exception occurs at the following line:

```
x[i] = mouseX/width;
```

He again consults HelpMeOut and sees a suggestion to surround the array access with an array bounds check (Figure 3). The suggestion also includes a plain text explanation of the problem and its solution. To indicate that he thought this particular suggestion was valuable, he clicks on the "vote up" link underneath the suggestion.

The explanation was provider earlier in the week by Jane, Tim's teacher, who was wondering how her students were doing. She visited the HelpMeOut web site and looked at a list of fixes that were frequently returned to other HelpMeOut users (Figure 4). She picked some of the suggestions and added explanations (Figure 5).

Figure 4. The HelpMeOut web interface provides a priority list of fixes that could benefit most from expert explanations.

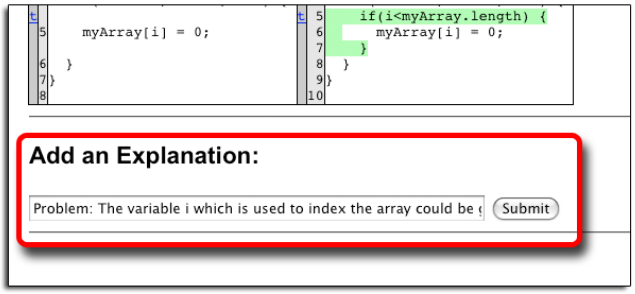

Figure 5. Expert users can provide explanations for a fix.

ARCHITECTURE AND IMPLEMENTATION

This section describes general techniques and algorithms for realizing crowdsourced debugging suggestions, and our particular implementation of these principles in the current HelpMeOut prototype.

We have implemented HelpMeOut for two programming languages popular with hobbyist and novice programmers so far. *Processing*[1] is a Java-based programming environment for multimedia and interactive graphics applications. It is popular as an introductory teaching tool. *Arduino*[2] is a programming environment for microcontrollers popular with creators of tangible interfaces and physical computing. The underlying language is a subset of C++. We will use the Processing/Java implementation as an example, then comment on differences between the two implementations.

Collecting Example Fixes Through Change Tracking

To automatically collect examples of errors and fixes, a tool has to keep track of both source changes and program status (compilation results or runtime errors) as the source is edited and run throughout a development session. HelpMeOut employs different strategies for collecting fixes for compiler errors and runtime exceptions.

Compiler Errors: What Changed to Make the Code Compile?

For compile time errors, a fix is a source change that takes a project from a failed compilation to a successful compilation. HelpMeOut monitors return codes from the Processing compiler throughout a programming session with a finite state machine (Figure 6). If compilation fails with an error, the error message and a snapshot of the source are saved. If the subsequent compilation succeeds, a

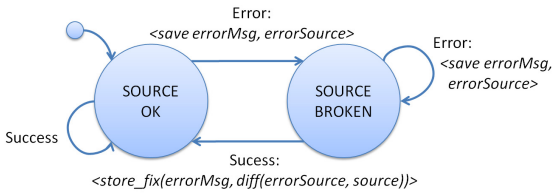

Figure 6. A state machine tracks compiler errors to collect fix reports for the HelpMeOut database. *Error* connotes a failed compilation, *Success* a successful compilation.

diff report [14] comparing the initial error state and the error-free state is generated. The error message and the diff report are then sent to the remote HelpMeOut database to be stored as a bug fix.

Runtime Exceptions: Did the Program Make Progress Past the Previous Point of Failure?

Automatically recording fixes for runtime exceptions is arguably more useful, but also harder. While it is easy to detect when a program is broken by watching for runtime exceptions during execution, it is not obvious when such a problem has been fixed. If a program had an error at a given line of code and runs successfully on the following execution, this could be attributable to either a successful bug fix; or no bug fix, but the bug not manifesting itself, *e.g.*, because of different program input.

While detecting whether a runtime bug has been fixed is undecidable in the general case, HelpMeOut employs a progress heuristic that catches a useful subset of exceptions. When a runtime exception occurs, HelpMeOut saves the error message, the stack trace, line number in the source file, and the number of times the line had been called when the exception occurred. On the following execution, a diff algorithm calculates the line in the new modified source that corresponds to the line where the exception occurred in the old source. The runtime system then counts the number of times this line gets executed. If the line execution count reaches the count of the previous error and the program subsequently makes progress, HelpMeOut marks the exception as resolved.

Progress tracking relies on an augmented Processing runtime system that can interpret Java code (instead of executing compiled code) to supply line execution counts. It would also be possible to achieve similar functionality by augmenting the Java Virtual Machine.

Finding Relevant Examples in a Database of Fixes

Whenever an error occurs in a programming session, due to a failed compilation or an exception, HelpMeOut generates a query to its remote database to retrieve related fixes, based on the error message as well as the line of code referenced by the error.

The database of example fixes has to be reachable from many individual users' machines, store submitted reports, and return related fixes in response to a query containing an error and code context. To achieve easy access, HelpMeOut implements the database as a web service that can be

[1] http://www.processing.org/

[2] http://www.arduino.cc/

queried through HTTP requests. In our prototype, we extend an Apache web server with Python CGI scripts to respond to remote procedure calls using the JSON-RPC[3] format. The database is implemented using SQLite[4].

Relevance matching follows a three step process:

1) Query existing fixes based on matching the error message from the compiler error or runtime exception.
2) Rank-order results from step 1 according to similarity of source structure or stack trace structure. Return m-best list.
3) Re-order results from step 2, based on previous user votes; Return n-best list, where n<m.

We next review the query process for compiler errors and runtime exceptions in detail. Our current algorithms are proof-of-concept implementations that are sufficient to test the HelpMeOut user experience. They can be improved upon with more robust approaches in future work.

Step 1: Matching Errors Messages

As a first step in identifying relevant fixes, the error messages of query and database entry have to match. Our current implementation checks for string matching with wildcards replacing identifiers and literals inside the error message, as these are likely to be unique to the user's program. For example, an error for "Unexpected token: myVar" generates a query for "Unexpected token: %", where % is the SQL wildcard character.

Step 2a: Determining Relevance for Compiler Errors

From the set of errors fixes obtained through matching of error messages, which fixes are most relevant? We hypothesize that *a fix is relevant if the source code of the broken state in the fix contains a line that is as close as possible to the line of source code referenced by the error in the query*.

A naïve approach for calculating similarity would be Levenshtein's string edit distance [21] between the two source lines. However, edit distance over source code overly penalizes changes in the identifier names, literals, and comments, which are likely to vary between different users' programs. We therefore employ a more robust approach in which source code is first passed through a lexical analyzer, which discards whitespace and replaces identifiers, literals, and comments with placeholders. An example of this tokenization is shown in Table 1.

Similarity between two lines of tokens is then calculated using a similarity ratio, where identical lines have a similarity of 1; lines that do not share any characters have a similarity of 0. We employ the Python difflib[5] ratio, which is $2 \times M/T$, where T is the total number of characters in both lines, and M is the number of matched characters according to a sequence differencing algorithm. The similarity for an entire fix, which may contain many lines of changed source, is then calculated as the maximum similarity encountered when comparing all lines individually against the input line. Alternative approaches for similarity detection from the literature on code clone detection, *e.g.*, parse tree matching [18], could be substituted.

Source	Tokenized Source
`/* a comment */`	`c`
`float[] x =new float[50];`	`float[]n=newfloat[il];`
`void setup() {`	`voidfn(){`
`  x[0]=1.0f;`	`n[il]=fl;`
`  smooth();`	`n();`
`}`	`}`
Substitutions in this example:	
comment = c, name=n, integer literal = il, float literal = fl, function name in definition=fn	

Table 1. Example of lexical source transformation performed during similarity calculation.

A subtle difficulty that will require more attention is that the line number reported by a compile error does not necessarily match the line where the real problem occurs. To hedge against this problem, analyzing an entire block of code surrounding the reported error line is advisable.

Step 2b: Determining Relevance for Runtime Exceptions

For runtime exceptions, we hypothesize that *a fix is relevant if as much as possible of the exception's stack trace in the user's query matches the stack trace of the broken code of the candidate fix in the database*. Exceptions are often raised by standard API methods, and similarity of the chain of calls from the user's code into the failing API method is indicative of similar intent across different programs. Because the highest levels of a stack trace are likely to be user-defined functions which will not match across programs, HelpMeOut calculates stack trace similarity as the number of consecutive shared lines starting from the bottom of the stack, *i.e.*, from the method that first threw the exception.

Step 3: Re-Ordering Based on User Votes

Since there is no editorial control in the bug fix collection process, variance in the utility of collected fixes should be expected. To promote fixes that users have deemed useful and to demote fixes that are not helpful, HelpMeOut includes functions for users to vote presented fixes up and down. Many approaches for factoring user feedback into selection algorithms exist. HelpMeOut retrieves 2N best examples and then reorders these examples in decreasing order of votes (each up vote = +1, down vote = -1). The best N fixes are then returned to the user.

Presenting Found Fixes

The list of relevant fixes generated in the previous step is visualized in a separate pane inside the programmer's IDE. The visualization juxtaposes before (with error) and after (without error) states of the code, and highlights what parts changed. Only changed lines are shown to conserve space.

[3] http://www.json-rpc.org

[4] http://www.sqlite.org

[5] http://docs.python.org/library/difflib.html

Below each code comparion are links to vote a given example up or down. When the user chooses to vote up or down, the vote is added to the database, which is then re-queried to immediately show new results. The voted fix may move towards the top of the list or further down, potentially dropping out of the top N list and being replaced with a different fix. The view limits display of code context for any given fix to converse space and show multiple possible suggestions. If the user needs more code context, a "more detail" link takes them to a web page that contains a full-file difference view in an external window.

Integrating fixes into user code

Once relevant examples are displayed, the remaining challenge is to determine whether the suggestions are applicable to the user's code and, if so, apply changes that fix the user's problem. These steps can be accomplished manually, automatically, or with mixed initiative.

HelpMeOut can attempt to automatically apply a suggestion to the user's program. This automatic patching is currently limited to single-line changes. HelpMeOut first tries to find the line where the fix should be applied (this is often not the line where the error occurred). Again, to avoid mismatches due to variable names and literals, source code and fix are tokenized. If a line was found, HelpMeOut then calculates a token-based diff between the fix and the user's source line. When the difference set is applied to the user's source, preference is given to the user's text for any matching tokens. This ensures that the user's variable names and values are preserved where possible (Figure 7). For multi-line patches or situates where automatic patching fails, HelpMeOut pastes the fix into the user's code as a comment so it can be integrated manually.

Augmenting Examples with Explanations

Presenting only examples may make the transfer from example code to user code challenging. Presenting a principle that explains how the example fix works can likely help. But where should these principles come from? Two options are generic explanations of error messages, *e.g.*, from the compiler documentation; or specific explanations of the error and its fix in the context of the given example.

HelpMeOut leverages an online community of users to provide the latter kind of explanations. HelpMeOut logs all

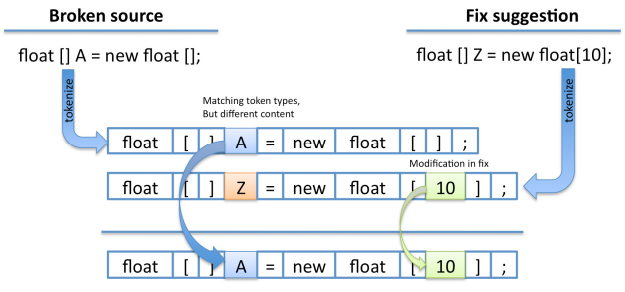

Figure 7. An example of token-based patching for automatically applying fixes to user programs.

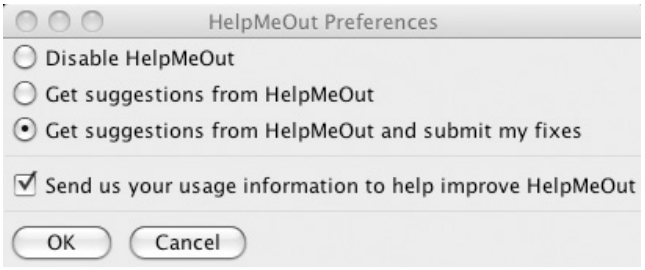

Figure 8. Privacy preferences in HelpMeOut give users control about exposing their code to others.

database queries so statistics which fixes are shown most frequently to users are available. Having explanations for those frequently returned fixes would be most useful. The HelpMeOut web interface presents a priority-ordered list of fixes that still need explanations so experts, *e.g.*, teachers, can browse these fixes and supply explanations.

Keeping Private Data Private

The need for users to keep all or parts of their code private may prevent them from using HelpMeOut. Setting privacy preferences can mitigate some of these concerns.

Preferences enable setting whether to query and submit fixes, query only (some code will be sent to the database, but it will not be visible to others users); or disable HelpMeOut (Figure 8). Independent of querying behavior, users can also choose to upload usage logs which contain command counts and error messages encountered, but no user code, to the database. A more detailed treatment of privacy questions is provided in the discussion section.

HelpMeOut For Other Programming Languages

To evaluate whether the functionality in the initial HelpMeOut Java implementation transfers into other domains, we ported its architecture to the Arduino development environment. Arduino and Processing share the same IDE code base, but target different compiler back ends: Arduino is used to write C/C++-code for microcontrollers; it relies on the open-source avr-gcc[6] compiler.

We noted that the gcc compiler generated error messages such as `"error: at this point in file"` that do not provide any information about the cause of the problem. Such error messages are a good example for the need for augmenting error message queries with source code context. While the (lack of) quality of error messages may make HelpMeOut more appealing for Arduino, HelpMeOut cannot capture or provide suggestions for any runtime errors because the compiled program is not run on the development machine itself, but on an external microcontroller.

This exercise led us to reconsider the language space for which techniques such as HelpMeOut have the largest potential impact. In future work we plan on supporting dynamic scripting languages such as JavaScript, Ruby, and Python. Such languages are frequently used by our target

[6] http://www.nongnu.org/avr-libc/

audience of amateur programmers. While many helpful static verification techniques are available for Java, tool support is comparatively low for dynamic languages.

EVALUATION

Our initial evaluation sought to establish evidence for the feasibility of the HelpMeOut end-to-end approach for collecting and displaying bug fix suggestions. Our evaluation considered the following three concrete questions:

1. Can we quantify, for our chosen IDE and language, how large the example set needs to be? How many examples and different users are needed before suggestions are returned for a majority of queries?
2. How *useful* are bug fix suggestions collected during instrumented programming sessions?
3. Which *types of errors* are covered well by HelpMeOut, which ones are not?

Method: Two Programming Workshops

We evaluated HelpMeOut through two three-hour workshops on Processing offered to graduate students at an Art & Design school in our area. Most students self-ranked as novice or "struggling" programmers with no or brief prior exposure to Processing (Figure 10). Students downloaded a version of Processing with HelpMeOut at the beginning of the first workshop and used it for both sessions. 8 students used HelpMeOut in the first session; 5 in the second. This resulted in approximately 39 person-hours of programming data. Students all worked on the same set of problems. Thus, our results are relevant for deployments in homogenous groups, *e.g.*, in a class or company, but may not be representative of highly heterogeneous user groups.

To seed the database with some initial fixes for common errors, we transcribed the examples in the debugging chapter of Shiffman's Processing textbook [29] as before/after source pairs and added them to the database. This set comprised 12 runtime fixes and 21 compile-time fixes.

Results

During the workshop, students queried HelpMeOut 274 times (7 queries per person, per hour). 229 queries (84%) returned at least one suggestion from HelpMeOut, meaning that at least one fix with a similar error message existed in the database at the time. This suggests that *common errors are common enough to have example fixes after relatively few hours of usage*. Whether these fixes are helpful will be addressed further below. 238 queries (87%) were for compiler errors; 36 for runtime errors. The dominance of compiler errors may be due to the format of the tutorial where students worked through a number of projects in fairly quick succession.

	Knowledge of Processing language	Programming Expertise	Interaction Design Expertise
No knowledge	4	2	1
Passing knowledge	3	4	4
Working knowledge	1	2	1
Fluent	0	0	2
Expert / Professional	0	0	0

Figure 10. Self-reported expertise of workshop participants.

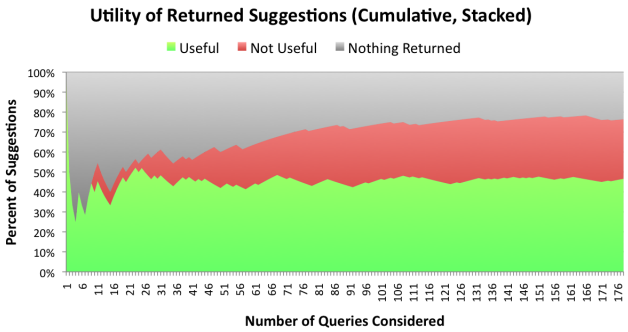

Figure 9. Relative utility of returned suggestions for queries issued during the Processing workshops.

Students submitted 101 fixes (2.6 per person, per hour, 88 compiler error fixes, 13 runtime fixes). Even within the relatively short time span of 39 person-hours, many of the fixes that were newly submitted were recycled and returned to other users (or the same user). In one example we observed, a student had a compile-time error and found out that the fix suggestion presented by HelpMeOut had been entered by his neighbor struggling with a similar problem just a few minutes earlier.

How useful are the returned suggestions?

We manually examined each query generated during the workshops and the suggested fixes returned at the time to determine utility of suggestions. We operationalized utility as follows: given the error message and the line of code reported as the error line, does at least one of the returned suggestions lead either to a direct solution of the problem or to a clarification of the problem that suggests a solution? One example of a direct solution is a syntax error where "}" was used instead of "]", and the fix suggests this exact substitution. An example of an indirectly useful suggestion is a misspelled function name where the suggestions show other misspellings that were corrected, but not for the same function name.

For 96 of the 274 student queries we could not determine whether the suggestions were helpful or not, mostly due to limited code context in our log files. We labeled the remaining 178 queries with three categories: helpful, not helpful, and no suggestions returned.

On average, for this data set, 47% of queries yielded useful suggestions, 25% were not useful, and 23% yielded no suggestions. Figure 9 shows how these percentages evolved over time. The percentage of queries for which no suggestions were returned decreases over time, as should be expected. However, the percentage of useful suggestions so far hovers consistently just below 50%. In other words, every other query returns useful suggestions. Why are useful results relatively steady? One possible explanation is that there are still many distinct error instances for a given error messages that we have not captured in the database yet. We would predict the rate of useful suggestions to eventually rise in this case. A larger deployment with more varied programming tasks and a larger dataset will have to

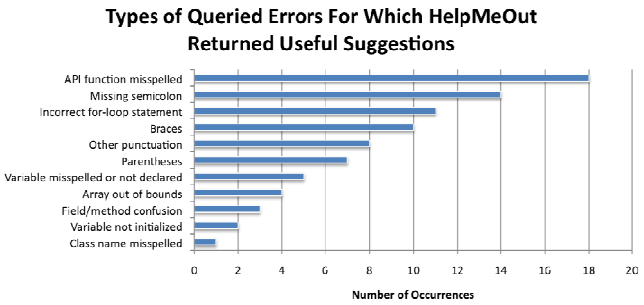

Figure 11. Error types for queries that yielded useful suggestions from HelpMeOut.

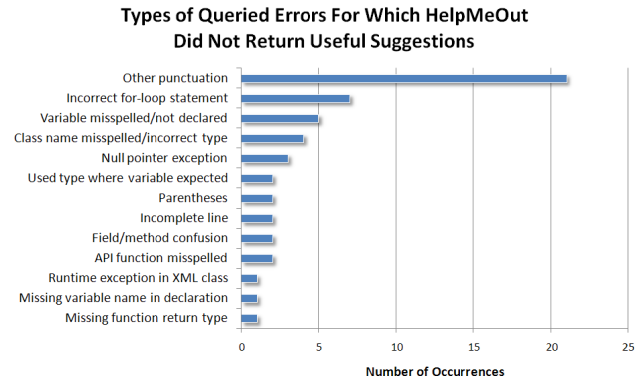

Figure 12. Error types for queries that yielded suggestions, but where suggestions were not useful.

determine to what extent utility can be increased. A second possible explanation is that the current relevance algorithm systematically fails for some subset of errors. Our analysis below suggests that this option is less likely.

What errors can be corrected?

Are there characteristic differences in the types of error queries that yielded useful versus non-useful suggestions? We manually categorized the errors contained in the queries that yielded useful suggestions (Figure 11) and those that did not yield useful suggestions (Figure 12).

What do these results suggest about the performance of HelpMeOut? First, the predominance of "miscellaneous" punctuation syntax errors in queries that did not produce useful suggestions points to the fact that misplaced punctuation can manifest itself in many different error types and different places in code. HelpMeOut's database, while covering the appropriate error messages, did not contain appropriate matching lines of source code yet. So a larger corpus of examples will be needed, and should be effective. Second, beyond the "head" of this distribution, there is also a longer "tail" of queries that did not yield useful suggestions: again, only more data will increase the likelihood of having seen less frequent error types.

The principal realization from this analysis is that a larger set of underlying causes map onto a smaller set of error messages. While HelpMeOut achieved 84% of coverage when only considering error messages, our data suggests that many underlying causes are not yet represented in the fix database. This suggests that the rate of useful fixes could in fact rise with a longer deployment of HelpMeOut.

We finally note that our analysis is inconclusive for runtime exceptions. For runtime errors, the line where the error manifests itself is frequently not the line where the problem has to be fixed. Because this remote error line was not captured in our logs, most runtime errors could not be included in this analysis. We leave this evaluation to future work.

Follow-Up Questions

The presented evaluation suggests that the algorithmic approach of HelpMeOut shows promise, while leaving room for improvement. However, we have not yet evaluated the efficacy of the presentation interface: Are non-expert users able to take suggestions rated as useful by experts, and transfer the fixes successfully into their own code? And more generally, does presenting examples of compiler errors aid programmer understanding of error messages? We also have not yet compared HelpMeOut directly to current status quo techniques of looking up compiler errors in documentation or using web search. Finally, future evaluation should clarify the impact that explanations have on transfer performance. Literature on analogical problem solving [11,12] suggests that programmers seeking to transfer a fix suggestion to their code will be aided by an explanation that states the principle that the fix demonstrates, but we have yet to measure this effect.

DISCUSSION

This section reviews technical and social limitations of the collaborative approach embodied in HelpMeOut and identifies areas for future work.

Privacy — Is Sharing Realistic?

Can we assume that people will be willing to share their (buggy) code freely with the world? Within the open source software community, code sharing is already an established practice and we foresee no obstacles in adoption of a tool like HelpMeOut for this user class. But only a fraction of code is written as open source. In general, tools like HelpMeOut will have to navigate issues of private and proprietary code. HelpMeOut already offers the option to disable submitting fixes to the database, or to disable all database traffic. Two other possible scenarios that maintain code confidentiality are restricted group deployments, and personal, read-only databases.

Restricted Group Deployments

Our evaluation suggests that smaller groups of users, such as a class of students, or a product team within an organization, can generate enough fixes to create a useful database. Thus an installation that operates within a smaller social group where code sharing within the group is permitted, but sharing outside of the group is not, can still be useful. An added benefit of a local installation is that people who work on related code are more likely to make (and correct) related errors.

Personal Read-Only Databases

Another option is to only collect fixes from a group of users who opts in to supply those fixes, but to let a larger group of users who do not wish to share their code benefit from the database. Because each query also transmits some amount of the user's code to the database to establish a match, some users may not want to issue remote queries. The database file itself could be located on the user's own machine and updated periodically, so no private information is ever relayed to a third party.

Keeping private data private, selectively

Even in the case where a user community generally agrees to share source code, some code within a project should remain private. Examples are passwords and API keys stored as plain text in source code. We propose to address this issue through source code annotations. If an annotation is found preceding a variable declaration, that variable's value could obfuscated before code is sent to the server. This places some burden on the developer to remember to label data as private, but enables fine-grained control.

Plagiarism and Learning

Debugging tools for non-experts can have a variety of goals: one goal could be to teach students how to form correct mental models of compilation and program execution. A different goal would be to simply eliminate programming errors, whether or not learning takes place ("just fix it"). These two goals can be in conflict. For example, when we demonstrated HelpMeOut to Computer Science teachers in our department, they remarked that use of HelpMeOut in a class context could lead to a "free rider problem" where students who procrastinate on an assignment benefit from fixes added to HelpMeOut by students who started earlier.

Our motivation for HelpMeOut was to aid non-experts who are not primarily evaluated on the originality of the code they produce, but who have to write code as part of their work. Hobbyists, electronic artists, web designers fit this description.

Limitations

The presented implementation of HelpMeOut has several important technical limitations:

1) A simplifying characteristic of the Processing compiler used in our prototype is that it is configured to only report a single error. This facilitates association of a given code change with a given error.
2) HelpMeOut does not currently deal with type systems of object-oriented languages. All user-defined types are considered identifiers and are abstracted away during queries. A more sophisticated implementation would take inheritance relationships into account.
3) Lexical analysis as a basis for relevance matching and patching outperforms matching plain text, but has its limits. For more accurate matching, HelpMeOut should analyze parse trees if such trees can be constructed.
4) The progress heuristic used to detect fixes to runtime exceptions has limitations: it cannot deal with different application input between runs.

Finally, the degree to which amateur programmers can reason about the transfer of fixes from one program to another is an important empirical question that requires further investigation.

RELATED WORK

HelpMeOut related to prior work in five areas: studies of novice programmers; systems for finding and correcting bugs; example-centric programming; better programming IDEs; and instrumented authoring environments.

Programming Errors of Novices

Debugging by novices has been well-studied in the Computer Science Education community. For a recent survey, see [26]; a recent multi-institutional study is reported in [10]. Nienaltowski et al. [28] studied how different styles of compiler error messages are understood by novice programmers, finding that additional detail is not necessarily helpful and suggesting that information placement and structuring are more important. Our research goal is complementary in that HelpMeOut strives to improve debugging performance without changing compiler messages. Ahmadzadeh et al. [1] studied patterns of compiler errors in novice users' code using instrumentation similar to ours — but their results were manually analyzed, while HelpMeOut uses them to generate suggestions automatically.

Finding and Correcting Bugs

Bug detection is an active research area in software engineering. Some projects have specifically investigated how to find and correct bugs and program errors based on data collected from a development team or a larger user base. Kim et al.'s BugMem [19] uses the version control history of large, long-running software projects to find project-specific bugs and suggest fixes. One interesting result is that bugs found by mining project histories are largely distinct from bugs found by static analysis techniques, suggesting that tools based on code-to-code comparison can effectively augment other formal techniques. DynaMine [24] similarly extracts recurring patterns of application-specific errors by data mining project revision histories.

Liblit et al. [22] proposed to automatically instrument application binaries to collect statistical data of runtime behavior during real-world software deployment. The statistics are aggregated on a central server where the developer can inspect them to find runtime bugs.

Other research and commercial systems have focused on supporting remote synchronous debugging, where multiple developers engage in a conversation around a shared view of program source [8] or runtime state [30]. Domingue and Mulholland's goal to "foster online debugging communities" is also congruent with our motivation [9]. They argue that there are no successful online debugging communities so far because communicating bugs through plain text

forum posts place too high a burden on programmers to describe and understand bugs. Research on collaboration in programming has mostly focused on the corporate setting, where small, geographically distributed teams of experts are the norm. For example, the Jazz [7] project augments the Eclipse development environment with team collaboration tools.

Ko's WhyLine [20] is notable for its focus on debugging as a human cognitive activity that can benefit from reframing the debugging task as posing and answering a set of "why" and "why not" questions.

Finding Relevant Examples

Recent work has examined how to aid programmers with finding relevant example code for programming libraries. These projects differ from HelpMeOut by focusing on finding working examples of new functionality that does not yet exist in the user's code, rather than suggesting solutions to problems in the user's code.

Brandt's BluePrint system [4] integrates search for code examples directly into the development environment. Assieme [16] introduced an augmented code search engine that combines documentation search results with code snippets of the relevant function in use. Jadeite [31] uses data mining of published code examples to improve the documentation of libraries, *e.g.*, by resizing the font used to display function names to show their relative call frequency in real-world code.

Better Editors

HelpMeOut aids debugging by relying on crowdsourced suggestions; an alternative approach is to improve the compiler or code editor. Many of the compile-time errors caught by HelpMeOut in our evaluation could also be prevented by smarter editors, though this is not generally true for runtime exceptions.

Structured or syntax-directed editors (*e.g.*, the Cornell Program Synthesizer [32]) make it impossible to create syntax errors in the first place. However, such editors increase the *viscosity*—the resistance to change—making experimentation harder. Relaxed edit-time grammars have been proposed as a solution to this problem [2].

A second strategy is to provide auto-completion during editing (*e.g.*, Microsoft IntelliSense) and error highlighting through background compilation (*e.g.*, as found in the Eclipse IDE). Such techniques match source code against formal descriptions of APIs and errors; HelpMeOut matches against real-world occurrences of errors. HelpMeOut can thus catch errors caused by incorrect use of API conventions. HelpMeOut also provides explanations of concrete examples of errors and fixes. Incremental compilation is only applicable to compiled languages. This reinforces our motivation to apply HelpMeOut to dynamic languages in future work.

A third path is to provide better compiler errors [3,17]. We see such research as complementary to our work.

Instrumented Authoring Environments

Prior research has investigated how to extract information from authoring application usage logs to inform usability evaluation and to guide application users.

Hilbert and Redmiles [15] published a survey of event trace recording methods to derive application usability data. Terry et al. instrumented an open source graphics program to collect usage information [33]. Usage logs are shared publicly on a website, a practice they term "open instrumentation". To provide a level of privacy, logs are partially anonymized and abstracted.

Linton and Schaefer [23] instrumented a Word processor to log command usage over time; based on log data, visualizations instruct users how to more effectively use the application. More recently, Matejka et al. improve upon Linton's results in CommunityCommands [25], a command recommendation system for complex creativity software such as AutoCAD. One goal of CommunityCommands is to suggest useful functions that users are not yet employing in the product to help them gain expertise.

Grabler et al. [13] generate tutorials in graphics software by recording demonstrations of an expert user and generalizing instructions from that history. We share with this research the strategy of automatically logging salient events during application use, as opposed to explicit revision management by the user. Our approach differs by logging changes to source code instead of command histories.

CONCLUSIONS AND FUTURE WORK

This paper presented *HelpMeOut*, a social recommender system that aids the debugging of error messages by suggesting solutions that other programmers have applied in the past. The main contribution of this paper is a *new strategy of collecting and presenting crowdsourced suggestions for programming errors inside an IDE*. We described the general architecture for such a system, two implementations, an initial evaluation and a discussion of the potential benefits and limitations vis-à-vis other approaches.

The fundamental technical insight enabling HelpMeOut is to *use both error messages and source code context in the capture and search for relevant fixes*. We believe that the general approach of automatically collecting usage data, aggregating data over many users, and then suggesting actions based on that data has wider applicability beyond the realm of programming errors. We also believe the approach can help users learn about API usage. We would also like to explore how to extend our approach beyond text programming languages into other media authoring tools.

One interesting question going forward is to what extent systems like HelpMeOut can combine automatic instrumentation, matching, and fixing algorithms with explicit user interaction.

REFERENCES

1. Ahmadzadeh, M., Elliman, D., and Higgins, C. An analysis of patterns of debugging among novice computer science students. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, ACM (2005), 84-88.
2. Birnbaum, B.E. and Goldman, K.J. Achieving Flexibility in Direct-Manipulation Programming Environments by Relaxing the Edit-Time Grammar. *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing*, IEEE Computer Society (2005), 259-266.
3. Boustani, N.E. and Hage, J. Improving type error messages for generic java. *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, ACM (2009), 131-140.
4. Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S.R. Example-Centric Programming: Integrating Web Search into the Development Environment. *Proceedings of CHI 2010*, (2010).
5. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., and Klemmer, S.R. Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover. *IEEE Software 26*, 5 (2009), 18-24.
6. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., and Klemmer, S.R. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. *Proceedings of the 27th international conference on Human factors in computing systems*, ACM (2009), 1589-1598.
7. Cheng, L., Souza, C.R.D., Hupfer, S., Patterson, J., and Ross, S. Building Collaboration into IDEs. *Queue 1*, 9 (2004), 40-50.
8. Dixon, P. pastebin - collaborative debugging tool. http://pastebin.com/.
9. Domingue, J. and Mulholland, P. Fostering debugging communities on the Web. *Communications of the ACM 40*, 4 (1997), 65-71.
10. Fitzgerald, S., Lewandowski, G., McCauley, R., et al. Debugging: Finding, Fixing and Flailing, a Multi-Institutional Study of Novice Debuggers. *Computer Science Education 18*, 2 (2008), 93-116.
11. Gick, M.L. and Holyoak, K.J. Analogical Problem Solving. *Cognitive Psychology 12*, 3 (1980), 306-55.
12. Gick, M.L. and Holyoak, K.J. Schema induction and analogical transfer. *Cognitive Psychology 15*, 1 (1983), 1-38.
13. Grabler, F., Agrawala, M., Li, W., Dontcheva, M., and Igarashi, T. Generating photo manipulation tutorials by demonstration. *ACM Transactions on Graphics 28*, 3 (2009), 1-9.
14. Heckel, P. A technique for isolating differences between files. *Communications of the ACM 21*, 4 (1978), 264-268.
15. Hilbert, D.M. and Redmiles, D.F. Extracting usability information from user interface events. *ACM Computing Surveys 32*, 4 (2000), 384-421.
16. Hoffmann, R., Fogarty, J., and Weld, D.S. Assieme: finding and leveraging implicit references in a web search interface for programmers. *Proceedings of the 20th annual ACM symposium on User interface software and technology*, ACM (2007), 13-22.
17. Jeffery, C.L. Generating LR syntax error messages from examples. *ACM Transactions on Programming Languages and Systems 25*, 5 (2003), 631-640.
18. Jiang, L., Misherghi, G., Su, Z., and Glondu, S. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. *Proceedings of the 29th international conference on Software Engineering*, IEEE (2007), 96-105.
19. Kim, S., Pan, K., and E. E. James Whitehead, J. Memories of bug fixes. *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, ACM (2006), 35-45.
20. Ko, A.J. and Myers, B.A. Debugging reinvented: asking and answering why and why not questions about program behavior. *Proceedings of the 30th international conference on Software engineering*, ACM (2008), 301-310.
21. Levenshtein, V.I. Binary codes capable of correcting deletions, insertions and reversals (in Russian). *Soviet Physics Doklady 10*, 8 (1966), 707-710.
22. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., and Jordan, M.I. Scalable statistical bug isolation. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ACM (2005), 15-26.
23. Linton, F. and Schaefer, H. Recommender Systems for Learning: Building User and Expert Models through Long-Term Observation of Application Use. *User Modeling and User-Adapted Interaction 10*, 2-3 (2000), 181-208.
24. Livshits, B. and Zimmermann, T. DynaMine: finding common error patterns by mining software revision histories. *SIGSOFT Software Engineering Notes 30*, 5 (2005), 296-305.
25. Matejka, J., Li, W., Grossman, T., and Fitzmaurice, G. CommunityCommands: command recommendations for software applications. *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, ACM (2009), 193-202.
26. McCauley, R., Fitzgerald, S., Lewandowski, G., et al. Debugging: A Review of the Literature from an Educational Perspective. *Computer Science Education 18*, 2 (2008).
27. Nardi, B. *A small matter of programming*. MIT Press, 1993.
28. Nienaltowski, M., Pedroni, M., and Meyer, B. Compiler error messages: what can help novices? *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, ACM (2008), 168-172.
29. Shiffman, D. *Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction*. Morgan Kaufmann, 2008.
30. Smith, R.B., Wolczko, M., and Ungar, D. From Kansas to Oz: collaborative debugging when a shared world breaks. *Communications of the ACM 40*, 4 (1997), 72-78.
31. Stylos, J., Faulring, A., Yang, Z., and Myers, B.A. Improving API Documentation Using API Usage Information. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC'09*, (2009).
32. Teitelbaum, T. and Reps, T. The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM 24*, 9 (1981), 563-573.
33. Terry, M., Kay, M., Vugt, B.V., Slack, B., and Park, T. Ingimp: introducing instrumentation to an end-user open source application. *Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, ACM (2008), 607-616.
34. Yeh, R.B., Paepcke, A., and Klemmer, S.R. Iterative design and evaluation of an event architecture for pen-and-paper interfaces. *Proceedings of the 21st annual ACM symposium on User interface software and technology*, ACM (2008), 111-120.