Expert Crowd Support Systems for Software Developers

YAN CHEN, STEVE ONEY, WALTER S. LASECKI, University of Michigan, Ann Arbor

1. INTRODUCTION

Software programming requires strong logical reasoning, proficiency in programming syntax, and a great deal of coding experience. In order to make software development more efficient, developers often seek programming support from various resources, such as the Web and other programmers [Hartmann et al. 2010]. However, our previous studies have shown that current support tools are limited for many of the types of requests developers would like to make, such as high-level advice, personalized help, or project-specific code segments [Chen et al. 2016].

Current IDE tools like Blueprint [Brandt et al. 2010] provide contextualized, easily-accessible, and on-demand support for developers, but are generally limited in the types of feedback they can provide (e.g., syntax error highlighting and function auto-complete) because the system cannot truly understand user queries or the context of the problem. Recruiting a programming helper can provide personalized support for specific requests, real time interactions and feedback, and even the ability to hand off sub-tasks for independent completion. However, in addition to the monetary price of hiring a freelancer, current hiring processes take significant time and preparation effort costs (interviews, onboarding, etc.). Furthermore, none of these helper solutions provide the in-context support that IDEs do, adding the need for an additional context switch to and from a developers workflow.

To overcome these limitations, we propose a new class of support systems for software developers to request on-demand help by expert crowds. Unlike automated IDE tools, expert crowds can provide high-level feedback about code by inferring and considering the end-user developer's intent. Unlike developer forums, on-demand developers can provide rapid responses and write project-specific code segments (which is often explicitly discouraged in developer forums). This paper describes three studies that help inform the design of on-demand developer support systems. We then discuss the implications draw from the results and conclude with design guidelines for future systems.

2. EXPLORATORY STUDIES

We conducted studies to inform design of each of the three main stages of help seeking: users forming a request, the worker response process, and users integrating responses. To minimize the affects of prior expertise between subjects, our studies used a new programming language that we generated from parts of multiple existing languages. In this section, we describe each of our studies.

2.1 Study 1: Developers' Input Methods

To find out what methods allow developers to make requests easily and quickly, we compared three request modalities for describing requests: 1) speak the request (Voice), 2) type the request (Text), and 3) choose the request from a fixed set of options (Multiple Choice). We also compared two context selectors for specifying a request's context: 1) select a segment of content (Highlight), and 2) point to one location in the content (Click). The multiple choice options are pre-selected from the types of request that we found in our previous studies [Chen et al. 2016], and the multiple location options are the most common programming language constructors.

	voice (average/s.d.)	text	multiple choice
highlight	10.99 / 2.61	45.46 / 25.29	14.97 / 13.11
click	14.02 / 3.65	37.49 / 23.14	27.83 / 18.00
without context selector	21.48 / 8.41	33.91 / 17.40	25.09 / 9.48

Table I. : Time to make a request for each condition (average and standard deviation).

Combining these request modalities and context selectors, we designed a 3×2 condition matrix for our experiment, with another set of conditions that has only the modalities with no context selectors. Given three programming tasks, participants could either speak, type, or select the given options to make a request if they need help with the tasks. We recruited two to three workers from Upwork for each condition, and recorded the duration and content of each request, and editor activity.

We observed that the Text requests took the longest time to make on average because the effort of typing and better constructing the requests. (Table I). Multiple Choice has very diverse results, which could take from a couple of seconds to half a minute. We found that this diversity is associated with the familiarity of the list of options for developers. The time spent on Voice requests is uniform with around 10 to 30 seconds on average. We observed that this is often because the audio requests are less prepared and more informal.

For different context selectors, we found that participants spent more time making requests in conditions with only requests than in conditions with context selectors because of the time spent adding context. Two annotators annotated each request with a set of rubric to compare which condition generated more clear and understandable requests. We found that nearly 30% of Multiple Choice requests were implied (with confident of $\kappa = 0.65$) and more than 40% of Multiple Choice requests were unclear (with confident of $\kappa = 0.83$). This makes sense because software development is a dynamic task that is impossible to have a finite list of options to capture all the possible requests. This suggested that when making requests during development, Multiple Choice mode is not as efficient as Text and Voice mode. Therefore, using Voice and Text would be better options to use to support request making in future systems. These support systems should optimize the trade-offs between speed of making a request and content of requests, and also add content highlighting as reference interaction in the final system.

2.2 Study 2: Helpers Response Methods

To better understand the trade-offs between different methods that helpers can use to respond to requests, we designed an experiment with four conditions to compare three most commonly used methods of response writing: 1) select a segment of the content and click a button to write annotation associated with it (add annotation), 2) write an explanation in a text box outside of the editor (add explanation), and 3) add inline code or comments (code inline).

Based on the common requests that participants made in Study 1, we developed three tasks (developer requests to respond to) and asked participants to respond to them using the documentation of the synthesized programming language. We hired 12 participants with three for each condition, recorded their performance and conducted a post-task interview.

Our major findings are shown in Table II where we computed the usage of each response format quantitatively and analyzed feedback from interview. Additionally, among the six participants who did not use a consistent response format during the study, four of them started with one method for task 1, and then used annotation for task two, and then switched back to the first method for task 3. Based on this pattern and participants' later reports, we found the usage of response format depends on the types of requests. For example, a general question is better to use explanation to respond and a request relates to a specific line of code is better to use annotation. Code inline is good for code request where one can explain the code in natural language along with his editing. The takeaway is that these

Study 2: developers' angle				
condition	# of requests that used this method among all sessions	# of subjects that used this method for all tasks	characteristics	observation
annotation	8	1	1. for specific code 2. for short response	request is informally written
explanation	18	3	1. for general questions 2. for both short&long response	request is informally written more contextual information
code inline	13	2	1. for code request 2. for complicated task	use both comments and code

Study 3: helpers' angle					
condition	time for integration	av. missed steps (of 3)	advantages	disadvantages	needs / design takeaways
annotation	> 10 min	0	1.no interruption 2.clear context and connection	1.only short response 2. # responses cannot scale	1.discourage long response 2. high visibility
explanation	> 10 min	0.5	1.no interruption 2. better for long response	1.no direct reference 2. more navigation and mapping effort	1.encourage to answer general request 2.minimize the navigation effort when visualizing
code inline	< 10 min	0	1. less navigation effort	1. code interruption	1. discourage large scale response 2. encourage code example

Table II. : Some pros and cons of three response formats from both helpers and developers' angle, and design takeaways.

three methods complement with each other. The trade-offs between them depends on the types of the requests and helpers preference.

2.3 Study 3: Developers Integration Methods

To understand these three response methods (annotation, explanation, and code inline) from developers' perspective, we observed how developers integrate the same responses written in different formats. We ran the same four-condition experiment with one multi-step task, which contains subtasks that would be better to represent in one of the three formats.

To make sure the response in different formats contain the same amount of information, we developed two rules that take the same response and converted it into different formats. When converting annotation and code inline to explanation, we use line number to specify the reference; and we added or annotate the explanation at the beginning of the codebase when converting explanation to code inline and annotation, respectively. We recruited eight participants with two for each condition, and we recorded their screen video when they performed the study, and conducted a post-task interview.

Participants mentioned pros and cons for different response methods. For example, annotation can strongly connect responses with content, which complements explanation format's flaw by reducing the process of mapping response to code. Additionally, it does not interrupt the original code base as code inline response would do. However, it loses its advantages when either the total number of it or the content length is scaled. Explanation format does not interrupt the original codebase, and it works well for both long and short responses. Compared to annotation and explanation, inline code allowed participants to finish the task more quickly on average. This is because participants did not miss tracking the responses.

We summarized the pros and cons for each response format and drew design implications from the results that facilitate future system development (Table II). When developing the integration part of the support systems, one should consider the speed of integration, correctness of responses, and understandability of responses as three major factors.

3. RELATED WORK

Many tools have enabled crowds to aid in complex tasks. Chorus [Lasecki et al. 2013] uses continuous crowdsourcing to enable on-demand conversational interaction by recruiting multiple workers for conversational interactions with users. Apparition [Lasecki et al. 2015] enables prototyping interactive systems in real-time by introducing self-coordination mechanism to reduce task conflict among workers. Latoza et al. [LaToza et al. 2014] developed CrowdCode, which decomposes programming into self-contained function-based microtasks. In CrowdCode, clients make requests to the crowd with self-written specifications of the desired function's purpose and signature. Crowd workers are then automatically assigned to these tasks. However, such a system is limited in how much it can reduce the end user's time expenditure, since the request authoring process requires a detailed problem specification. Having the crowd to efficiently assist programming has been challenging in terms of qualified worker recruitment and efficient work evaluation. Collabode [Goldman et al. 2011] allows users to define function-based microtasks, including testing and debugging a function, which allows workers to evaluate the previous work. It also provides workers the choice of skipping the microtasks that do not match with workers' skill sets. However, the tool does not grant helpers the ability to choose the tasks that fit the best for them, and also the developers should assess the response instead of other workers.

4. CONCLUSION

In this paper, we proposed a new class of support systems for software developers to request on-demand help by expert crowds. Additionally, we described three studies that investigated the trade-offs between different methods on different stages of help seeking. We then draw design guidelines to facilitate future systems development.

REFERENCES

Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 513–522.

Yan Chen, Steve Oney, and Walter Lasecki. 2016. Towards providing on-demand expert support for software developers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM.

Max Goldman, Greg Little, and Robert C Miller. 2011. Real-time collaborative coding in a web IDE. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 155–164.

Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1019–1028.

Walter S Lasecki, Juho Kim, Nick Rafter, Onkur Sen, Jeffrey P Bigham, and Michael S Bernstein. 2015. Apparition: Crowdsourced user interfaces that come To life as you sketch them. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 1925–1934.

Walter S Lasecki, Rachel Wesley, Jeffrey Nichols, Anand Kulkarni, James F Allen, and Jeffrey P Bigham. 2013. Chorus: a crowd-powered conversational assistant. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. ACM, 151–162.

Thomas D LaToza, W Ben Towne, Christian M Adriano, and André Van Der Hoek. 2014. Microtask programming: Building software with a crowd. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM, 43–54.