Croder: Bringing the Knowledge of the Crowds into the IDE

Semih Okur
University of Illinois at Urbana-Champaign
Urbana, IL
okur2@illinois.edu

Mihai Codoban
University of Illinois at Urbana-Champaign
Urbana, IL
codo@illinois.edu

Caius Brindescu
University of Illinois at Urbana-Champaign
Urbana, IL
brind@illinois.edu

Kyungho Lee
University of Illinois at Urbana-Champaign
Urbana, IL
klee141@illinois.edu

Shuo Yuan
University of Illinois at Urbana-Champaign
Urbana, IL
syuan20@illinois.edu

INTRODUCTION

WHY INTEGRATE REVIEWS INTO THE IDE

Interruptions are bad because they break the flow of work and cause more stress [4, 3]. Also, the interruptions that a developers faces are varied [1], so trying to reduce that number will bring benefit to the workflow. By integrating the code review submission process into the *IDE* we can reduce the time of the interruption. Also, because the developer never leaves the context of his work, we hope that the effects of this interruption will be minimal.

Integrating into the *IDE* has other benefits too. The *IDE* is the environment where the code lives. By taping into this resource we can get a lot more information about what the developer what to review. For example, suppose he wants to select a method for review. If that method is tightly couple with another one, than we can suggest that he include that method as well. Also, the developer never leaves the context. He can very easily go back to code to decide whether some snippet of code would be relevant or not to the reviewer.

Another advantage is that we can submit discrete notification when a review has been received. This, again, reduces the distraction and makes it a lot easier to see what the review is about. By presenting the result of the review in the *IDE* the developer can see it in context. This makes it a lot easier to come back in the context of the review and make sense of the comments.

RELATED WORK

SOFTWARE DEVELOPMENT FOR CROWDS

The main research theme that was explored throughout this project is whether software development is suitable for the crowds. *Are there any software development micro-tasks which can be successfully accomplished in reasonably short time by the crowds?*

These tasks would need to take into consideration the main issue with crowdsourced tasks, that is, the lack of context. By the very nature of these short lived tasks workers cannot know the scope of the entire project. This poses severe limitations on task size since they have to be complete enough to offer the minimum amount of context for the work to be tractable, but must not be so large as to exceed the definition of micro-tasks. Deviations in any of the two conflicting constraints would make the task unattractive to workers.

We identified three activities that have a high chance of being crowdsourced.

Code reviews. In a usual code review, the author of the code selects a list of code snippets that he wants reviewed and then sends them to a list of reviewers. Reviewers usually consist of people working on the same project, since they have the most knowledge about the system particularities. They spend time going through the code both in isolation and together in a meeting. A list of issues is compiled at the end of the meeting. The usual outcomes of a code review are defect detection, design and code improvement, alternative solutions and dissemination of knowledge inside the team.

Therefore code reviews seem to represent a natural choice for crowdsourcing since they are usually performed by a team of people and the outcome represents the accumulated knowledge of these individuals. However, in terms of review outcomes, crowdsourced code reviews would provide value only to those outcomes that do not require great context. This means that tasks such as defect detection, high level design improvements and general issues that deal with the mapping of requirements to code would not pose good candidates.

On the other hand, crowds could be used to catch trivial and beginner mistakes. By deferring these issues, more value can be obtained from inside code reviews. Project colleagues would not waste time on these simple issues any more and could better dedicate their time and knowledge to finding high level issues that involve knowledge of the problem domain and implementation history.

Tests. The crowd could be asked to write tests for different software entities such as methods or classes. Such a task could prove suitable since there are standardized, almost mechanical procedures to writing tests. For methods, the crowd could provide black box tests, by making use of the method contract, or white box tests, by making use of the method's control and data flow. For classes, the crowd would create a composition of method tests together with class contract tests.

However, there are drawbacks and limitations to this kind of task. Tests would require some documentation of method and class contracts. The code under test may have dependencies that require to be stubbed away. High level requirement validation tests cannot be produced due to lack of context.

Code transformations on request. There are a series of tedious program transformations, both refactorings and behaviour changing transformations that are tedious enough to warrant automation but require too much domain knowledge to warrant automation. On one hand, developer time would be wasted on these mechanical tasks. On the other hand, the effort required to automatize them makes it intractable to apply to many, short lived transformations.

Examples on the refactoring side could be loop transformations for parallel execution, task boundary identification for parallel execution, code rewriting for better readability, etc. On the behaviour changing side, we can remind transformations such as introducing the visitor pattern to a large hierarchy or changing the code from using a certain library to another.

There are of course drawbacks and difficulties with this approach. Some tasks could require much more context information than the requester anticipated. The requester will have to inspect all proposed code changes which would annihilate part of the advantage of deferring work to somebody else.

From these three candidates for software development microtasks we chose to dedicate our efforts on providing support for code reviews. This seems like a natural starting point due to its social characteristics. From the three potential tasks, code reviews require the least amount of specialized, focused knowledge which should make them attractive to a wider audience and could potentially be of most value to the programmer. The latter justification pertains to the fact that tests and code transformations are of modest educational value to the programmer while code reviews aid in self improvement.

CROWDS FOR SOFTWARE DEVELOPMENT

One of the main challenges was finding the appropriate crowd to conduct the code review. One of the first options was the Amazon Mechanical Turk. The main problem with this platform is the lack of qualified workers. Code Reviewing is a very technical process that requires a large amount of knowledge. We needed to aim for a platform where we were guaranteed to have the right audience.

Mechanical Turk tasks tend to be very simple and require only minimal knowledge and cognitive skills. During one experiment we asked a technical question about JavaScript. Out of the 10 hits, 9 were complete in 7 days. Of those 9 tasks, only 4 useful and most of them were incomplete. This partly shows that the Mechanical Turk platform is ill-suited for tasks that require specialized knowledge.

Services such as eLance[1] and oDesk[2] employ a crowd to complete programming tasks. But unlike typical crowd sourcing platforms, is it an offer based system. The requester posts the description for a task and workers bid to complete it. The requester then chooses a winner and then work on the project can start. This system is not what we are looking for. We needed a system where you can post your task and workers would select the task and complete it for a predetermined amount.

StackOverflow[3] allow uses to post questions and get answers. The service is larger than most social Q&A and technical forums. With a median answer time of 11 minutes and a very active user user base [2] it makes a good candidate for a platform to run the experiment. StackOverflow is part of a network of sites (StackExchange[4] that follow the same modes. One of them, *Code Review Stack Exchange*[5] is based around the concept of concept of code review. It is this platform that we have used to test our protype.

STACKEXCHANGE CODE REVIEW CHARACTERISTICS

INTERFACE WITH STACKEXCHANGE

SELECTING CODE SNIPPETS

The first step that the programmer must undergo when composing a code review is of course choosing the code snippets that she desires to be reviewed. The programmer must be able to compose several code snippets that together paint an overall picture of the concept they want to portray. For example snippets could range from lines of code, to loops, to whole classes and packages.

CRODER allows the programmer to easily choose snippets from many places in the *IDE* . The reasoning behind this is that if a resource contains or points to code, CRODER should be able to transform it into a snippet. Figure 1 shows various resources which CRODER can convert to code snippets, such as random editor selections, fields, methods and whole classes. As can be seen, resources originate from diverse views.

In order to track what snippets have been added to the review so far, a view is provided that holds each snippet. Figure 2 illustrates this concept.

With today's CRODER the decision on the code snippets that are to be reviewed falls solely on the responsibility of the programmer. The Future Work section describes a technique which offers suggestions on other possible snippets based on the current ones.

CROWD SOURCED PEER REVIEW CREATION

[1] `http://www.elance.com`
[2] `http://www.odesk.com`
[3] `http://www.stackoverflow.com`
[4] `http://www.stackexchange.com`
[5] `http://codereview.stackexchange.com`

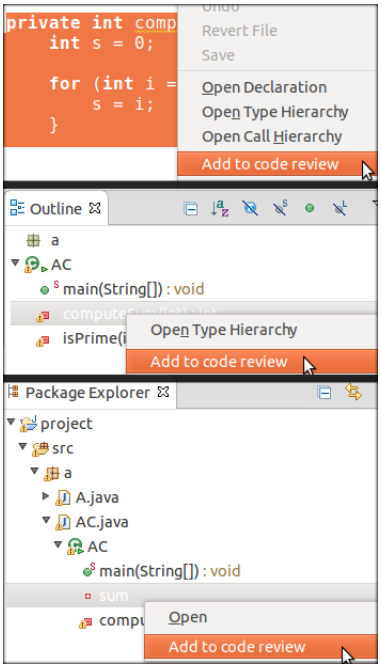

Figure 1. Adding code snippets from various resources

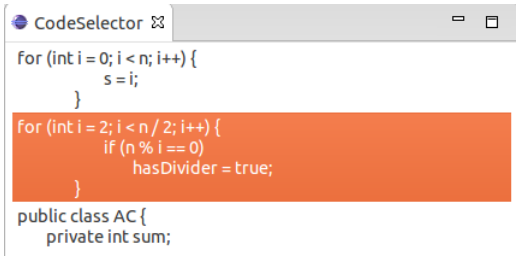

Figure 2. Viewing code snippets

For creating of the task we chose to implement a wizard. One of the main features is that you can ask for *structured* code review. The user can select a criteria under which code review should be performed (performance, design, readability etc.). This helps reviews in keeping a focus on a given task. Also, to make it easier for them to understand the code, we allow the users to add a small note regarding the purpose and content of each code snippet. The code snippets are selected directly from the IDE.

In figure 3 we show the design approach we took. While it is a bit crude, it does offer all the features and presents the structured approach to creating the task. It is worth mentioning that the user has the option to add a general comment at an earlier stage in the wizard.

After entering all the details needed for a task, the user can then select the service to post it to. For the moment we only offer integration with StackExchange, but other services can be added as well (like oDesk, eLance etc).

TYING REVIEW OUTCOMES TO THE CODE

Once you submit the code, it can be difficult to remember what task relates to what code. In order to help the programmer we decided to mark code snippets that were sent off for review. The icon on the right rules notifies the developer if any replies have been received.

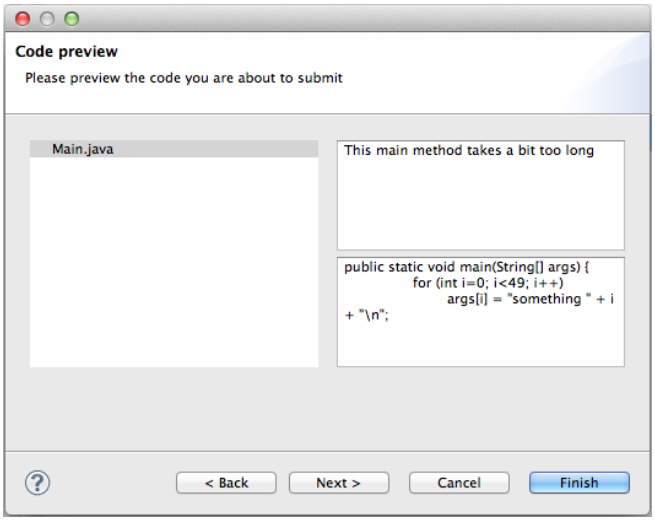

Figure 3. Adding comments for each snippet

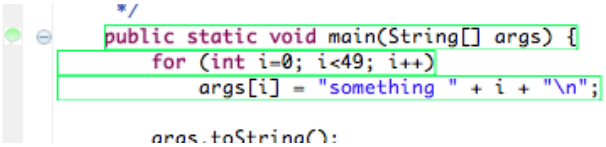

Figure 4. The market that corresponds to the review

Figure 4 presents this concept. The code marked by the red box has been submitted of review. Clicking on the marker will bring up the review task associated with it.

REVIEW MANAGEMENT IN THE IDE

In time, the programmer creates many reviews as she seeks to improve herself and her code. She may also wish to keep certain reviews for future reference.

CRODER keeps track of current and past reviews. Moreover, for each review it fetches the associated replies. Figure 5 illustrates the Review Browser. The left pane shows the review titles while the right pane shows the replies for the currently selected review.

Section illustrates several features which can enhance the tool's capabilities via review management.

PRELIMINARY USER STUDY

SKETCHES

Based on user interview study, we start to draw some initial sketches of workflow. At the beginning of drawing sketches, we focus on interface simplification. Many interviewee mentioned that it is a really hard time at the beginning of learning code specially for designers, architects or other students – because they do not have any professional education background of computer science or coding. So based on simplification principle, we hide many features on main menu, make

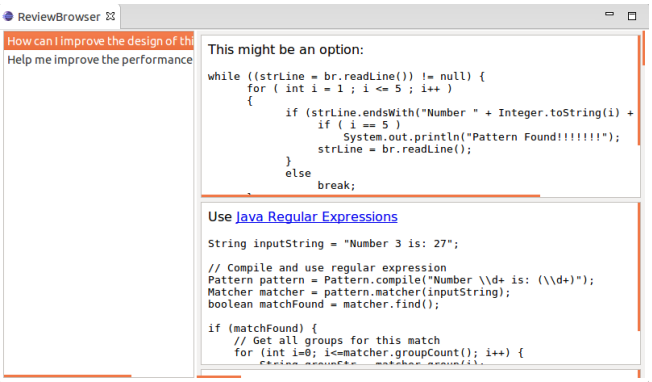

Figure 5. Viewing code snippets

it more user-friendly and still have consistency and hierarchy between different features (see Figure 6).

After Showing these initial sketches to our interviewees, we got feedback about layout and interface graphic hierarchy. Based on these feedbacks, we developed a new interface for our program (Figure **??**). Using metaphor method, the interface is like a hand-made leather cover notebook. This interface can recall users memory of using notebook to take note in class, in the meantime, make first time user specially user without background of coding quickly familiar with program.

FUTURE WORK

Future work can be done in several directions.

The only platform currently used by CRODER is Stackexchange code review due to reasons such as familiarity with the platform, constant activity and most importantly, it represents a crowd that is accustomed to the code review task type. Future work in this regard would see CRODER gain adapters for other platforms such as eLance or oDesk. On one hand, these platforms have mature APIs and sandboxed testing environments. On the other hand their crowds are not accustomed to micro-tasks.

Indeed, as was stated in section , eLance and oDesk crowds perform software outsourcing. Their tasks range from days to months and the compensation from tens to thousands of dollars. If we are to utilize these crowds then we shall have to experiment and see if we can get them accustomed to the micro-tasks of crowdsourcing. This implies that we will have to effectively teach the crowd. We shall have to start with oversized tasks and progressively decrease them in size and scope. We shall have to experiment with task size and compensation amount in order to find the sweet spots.

Automatic evaluation of crowdsourced reviews is also problematic. One cannot simply generate random code snippets of different size and complexity. Code snippets have to have a coherent story behind them otherwise the required specialized workers will not engage in their review.

Relating to code snippets, CRODER currently leaves code snippet selection solely in the hands of the programmer. Unfortunately, while collecting the code to be reviewed, the programmer may miss essential snippets that complete the story

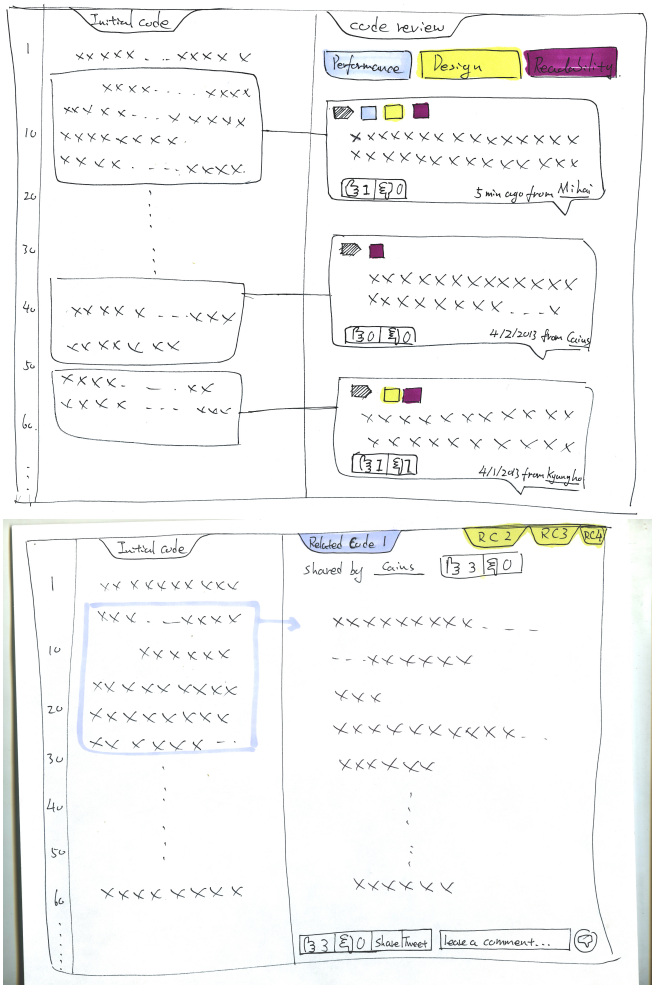

Figure 6. Initial design sketches

of the review. We propose a semi automated approach in which CRODER would suggest a least amount of additional snippets in order to have a more complete review. There are different solutions we can utilize in order to achieve this goal such as a metric based approach: a dependency and coupling analysis can be performed on the selected code snippets in order to find outside potential snippets that would minimize certain metrics.

In terms of the current implementation, particular code reviews are tied to only one *IDE* installation. The feature to share and access the review results with other colleague's *IDE* 's would be most welcome.

Last but not least, code reviews represent only one possible software development task that can be crowdsourced. We identified several more, such as testing or on demand code transformations. In order to fully bring the knowledge of the crowds into the *IDE* , CRODER will have to learn how to crowdsource many more tasks, starting with the ones previously mentioned.

CONCLUSION

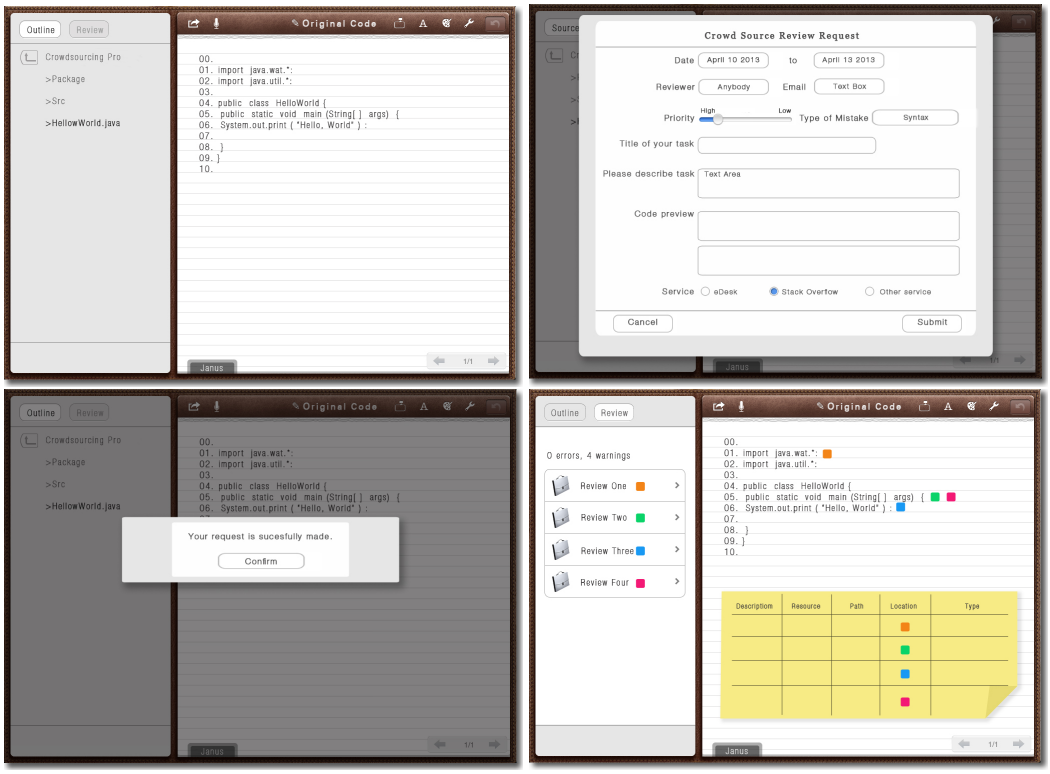

Figure 7. Interface example

REFERENCES

1. Czerwinski, M., Horvitz, E., and Wilhite, S. A diary study of task switching and interruptions. In *Proceedings of ACM CHI 2004 Conference on Human Factors in Computing Systems*, vol. 6, ACM Press (New York, New York, USA, 2004), 175–182.
2. Mamykina, L., Manoim, B., and Mittal, M. Design lessons from the fastest q&a site in the west. *Proceedings of the . . .* (2011), 2857.
3. Mark, G., Gudith, D., and Klocke, U. The cost of interrupted work: more speed and stress. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2008), 107–110.
4. Spolsky, J. Human Task Switches Considered Harmful.