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

ABSTRACT

This paper proposes a new way of doing code reviews. Since they take up a lot of time and require a great amount of coordination, we suggest using the crowds to fulfill this task. By integrating the *IDE* with the review platform we hope to make it easier for developers to review code and have code reviews. To test our proposed solution, we have built a new tool, CRODER, that can post reviews and get results from StackExchange. This integration will help developers post tasks without the need of context switching.

INTRODUCTION

Crowdsourced computing tasks have been sprouting everywhere lately, spanning over a diverse field of task types. They range from demographic studies (surveys), to personal inquiries (product preferences), to machine learning algorithm training (categorizing photos) and so on and so forth. One is limited only by imagination when customizing tasks for the crowd.

This spike in crowdsourced work has been instigated by the rise of supporting platforms as well. Mechanical Turk[1] has risen to be the *de facto* crowdsourcing platform. The large worker base that this platform has enables a high degree of parallel task decomposition. Moreover, the same large worker base enables rapid task response times. By means of these two crucial characteristics crowdsourcing has become a viable means towards task execution and decomposition.

An interesting application for crowdsourcing is in collaborative design. Workers with or without specialized knowledge participate to create something with a higher scope than the individual task. It falls to the requester's creativity to build tasks in such a way that by composing their results a bigger, holistic product is achieved.

Starting from the previous observation and from related work as well we decided to explore the space of crowdsourced software development tasks. In this regard we cross-referenced the particularities of the crowd with the particularities of software development tasks in order to find suitable candidates. We produced three potential tasks that would be suitable for crowdsourcing: code reviews, test generation and semi-mechanical code transformations.

For the scope of this paper we choosed to focus on code reviews for reasons detailed in section Software development for crowds. In this regard we built a tool, CRODER, which integrates with *Eclipse* to provide crowdsourced code reviews directly in the *IDE*. With CRODER, programmers can build and run the whole process of code reviews directly from the *IDE*, from selecting code snippets, to posting the review to the crowd, to managing and accessing review outcomes.

In terms of crowd platforms we could not use Mechanical Turk because of the lack of specialized crowds. Instead, we analysed potential replacements like eLance, oDesc and Stackexchange. For reasons further in the paper we chose Stackexchange as the backend of CRODER.

To evaluate the need for CRODER we conducted a study to assess potential user traits and to assess programmer's knowledge of crowdsourcing and code reviews. By means of this study we were then able to build three potential user personas and intuitive UI sketches.

RELATED WORK

Why Users Do Code Reviews Code review is a systematic examination of source code with an intention to find and fix bugs. Code reviews are primarily done during development phase to improve the quality of the software. The cost of fixing bugs after release to customer is nearly ten times the cost of the fixing it during development. Finding and fixing bugs early in the software development phase can lead to large cost savings [2].

Most code reviews are done by team members or by an automated source code static analysis tool. Automated static anal-

[1] `https://www.mturk.com/mturk/`

ysis tools can catch simple bugs based on predefined rules and are quite limited in terms of type of rules that can be defined. Peer code reviews done by team members are more useful in that they can identify much more impactful bugs, like implementation, multi-threading, boundary conditions, algorithmic and logic bugs, which can have higher impact on the software products quality and user acceptance.

Therefore, the code review tool should allow team members to review the code collaboratively in an easy and efficient manner. It provides all the benefits of formal code inspections and requires considerably less effort and time when compared with manual formal code inspections. Generally code reviews are done in a collaborative fashion. Code review tools facilitate the code review process by helping author and reviewers to review code effortlessly in a collaborative fashion. This makes team members more agile to work across multiple features of a project. Code reviews can be an effective mentoring tool for new team members. Code reviews help in shared learning. [6]. Code reviews fall into two main categories: Formal code reviews and Lightweight code reviews [11].

Formal code reviews involve a careful and detailed process with multiple participants and multiple phases. Formal code reviews are the traditional method of review, in which reviewers attend a series of meetings and review code line by line, usually using printed copies of the material. Formal inspections are extremely thorough and have been proven effective at finding defects in the code under review. Manual formal code reviews are quite laborious and generally require common time from all the participants of the review. Multiple participants are needed for review to play various different roles like Author, Reader, Tester, Moderator and Scribe.

Lightweight code reviews typically requires less overhead than formal code inspections, though it can be equally effective when done properly. Lightweight reviews are often conducted as part of the normal development process. Following are the different types of lightweight code reviews [7].

AVAILABLE COLLABORATIVE CODE REVIEW TOOLS

Considering the importance of code reviews, it has been often considered to be a complex, time consuming and highly regulated process as many developers conduct a code review. However, with the change of programming paradigms such as extreme programming, agile programming, the concept of code review has been transforming from formal code reviews to the lightweight, the agile way. Developers have become to realize the code process as a form of communication between team members, and they have tried to support tools that enable them to perform a code review in an efficient and collaborative way. Here are several code review tools to consider. [4].

Gerrit is a web-based collaborative code review tool for GIT. Starting from a set of patches for Rietveld, it became a fork and evolved into a full blown project when ACL patches wouldn't be merged into Rietveld by its author, Guido van Rossum. Originally written in Python like Rietveld, it is now written in Java (JEE Servlet) with SQL since version 2.

Review Board. An alternative to Gerrit, Review Board integrates with Bazaar, ClearCase, CVS, Git, Mercurial, Perforce, and Subversion. Review Board can be installed on any server running Apache or lighttpd and is free for both personal and commercial use.

Crucible is a collaborative code review application by the Australian software company Atlassian. Crucible is a Web-based application primarily aimed at the enterprise world, and certain features that enable peer review of a codebase may be considered enterprise social software. Crucible is particularly tailored to distributed teams, and facilitates asynchronous review and commenting on code. Crucible also integrates with popular source control tools, such as Git and Subversion. Crucible is not open source, but customers are allowed to view and modify the code for their own use.

WHY INTEGRATE REVIEWS INTO THE IDE

Interruptions are bad because they break the flow of work and cause more stress [12, 9]. Also, the interruptions that a developers face are varied [3], so a solution that reduces this number is essential. [1]. By integrating the code review submission process into the *IDE* we can reduce the impact 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 she wants to select a method for review. If that method is tightly coupled with another one, than we can suggest that he includes that method as well. Also, the developer never leaves the context. He can very easily go back to the code to decide whether some snippet of code would be relevant or not to the reviewer.

Another advantage is that we can use discrete notifications 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. We are hoping that this will increase the developer's productivity.

QUESTIONS FOR COLLABORATIVE CODE REVIEW

Based on the initial investigation on the code review itself and supportive tools, we can get valuable insights for defining the fundamental goals and its benefits in terms of a creativity support tools and crowd sourcing. Major benefits users could acquire through a code review are: [10]

- An extra set of eyes: catch bugs, mistakes or typos early while they are the cheapest to fix.
- Disseminate knowledge: Know colleague's code and prevent duplication by learning what others are working on.
- Keep things consistent: Make sure that code follows team guidelines.

- Improve user skills and knowledge: Learn from your team and grow as a developer.

With these benefits, we form three initial questions that are able to leverage those benefits in terms of creativity support tools and crowd sourcing. It is a part of the contextual inquiry that we took by planning interview questions.

1. Creativity support tools perspective. How can we design a creativity support tool for users to realize their bi-creativity through a code review process not only for overcoming technical challenges but also utilizing it as a learning opportunity? [4] [5]
2. Enhancing the crowd's performance and shepherding perspective. How can we design a tool that enable both of Requester and Workers to understand tasks and make them to focus on given tasks so that they make a good collaboration. What are effective ways to make Requesters and Workers to make a good collaboration even if they are not fully aware of the concept of the crowd computing.
3. Crowd critics and learning from the crowd's perspective. How can we design a tool that support users' quality of work significantly by motivating from crowds; what are effective ways to find qualified crowds and get insight from their experience?

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 specialized 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 have 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 number 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 separately 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.

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 methods that tests the class' interface.

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, 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 and task boundary identification for parallel execution, code rewriting for better readability, etc. On the behavior 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 micro-tasks 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 platform. 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 completed in 7 days. Of those 9 tasks, only 4 were 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[2] and oDesk[3] 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[4] allow users 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 [8] it makes a good candidate for a platform to run the experiments. StackOverflow is part of a network of sites, (StackExchange[5] that follow the same modes. One of them, *Code Review Stack Exchange*[6] is based around the concept of code review. It is this platform that we have used to test our prototype.

CONTEXTUAL INQUIRY

We conduct a contextual inquiry with 14 participants with different levels of programming experience, educational background and interests. Each interview and observation has taken about an hour or so in his or her workplace so the researcher could get insights in a more intuitive way. No incentives were given, all participants put their effort to this interview as a volunteers.

Interview Phase 1: Background Information

Most of participants that we interviewed have more than 4 years of programming experience and their preferred programming languages are JAVA, Javascript and Python. All of them hold a strong grasp of programming in their field.

In addition, most of them prefer using Eclipse and simple text editors such as Vim, Textmate, Editplus and Microsoft Visual Studio as an Integrated Development Environment (IDE). The major reason for *IDE* adoption is that many people choose and use it. Also, the other reason is that it is required to use with a certain language.

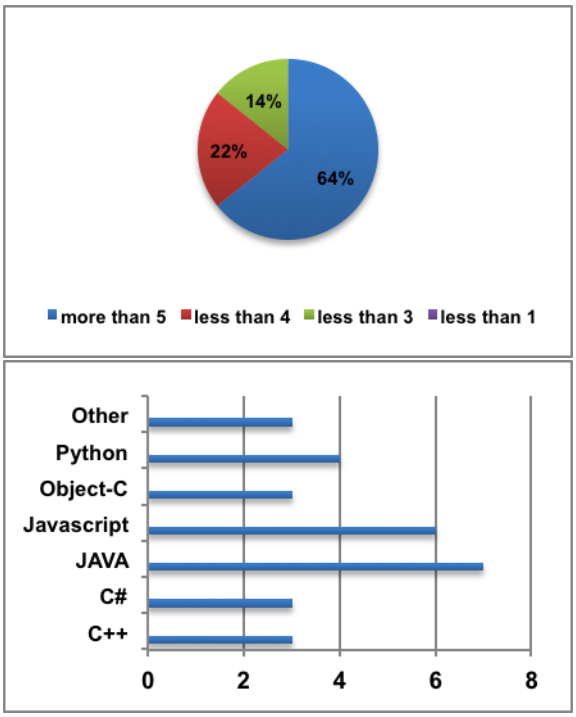

Figure 1. Participant programming experience and their preferred languages

Interview phase 2: Code review

Most of interviewees mentioned that their favourite developer community is Stackoverflow. The reason consists in its question database and active users, the statistics for getting good and fast answer being 72% and 14% respectively. They preferred having a code review from an expert who has much more knowledge on programming languages than themselves by 77%. Also, their main purpose to get a code review is to find a minor bug and test more cases 36% and to improve their programming capabilities 36%.

Specifically, they are interested in solving the following 5 problems through a code review:

1. Unused code (functions, variables, constants, etc.)
2. Inefficient code (misuse of dynamic filters, heavy constructors, etc.)
3. Over-complex code (nested loops, too many conditionals, etc.)
4. Over-long code (classes, methods, etc.)
5. Incorrect use of the language component or library lifecycle

Interestingly, 43% of interviewees answered that they want to get feedback from reviewer in an indirect way, mostly by annotating or email (indirectly) 43% or commenting in between source codes (indirectly) 36%.

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

Figure 2 shows the preferred *IDE* and reasons. Answer A represents "Because it is essentially related with the language" and B represents "Because many people choose and use it".

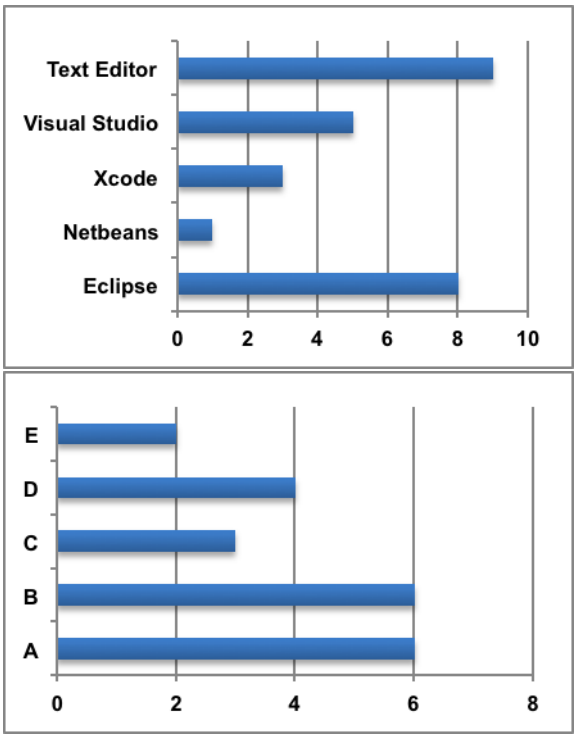

Figure 2. Most preferred IDE (left) and reason (right)

.

Interview phase 3: Code review through Crowd sourcing

Participants are not familiar with using crowd sourcing as it is an emerging technology in terms of code review. However, more than a half of interviewees (57%) mentioned that they know the concept, but just haven't used it so far (29%).

PERSONA

By analysing the results from our interview and observation, we create several persona so that we can understand the user's situation and needs. the users' voice is like this: "I need some expert who can help me get a code review, I preferred to have it in an indirect way. But I couldn't. There's no expert in Mechanical Turk!" "I really want to have qualified answers with code snippets and I also want to improve my capabilities". The persona traits can be found in table 4

DESIGN GOALS AND MAJOR FEATURES

Through the contextual inquiry, we could narrow down our initial questions and problematic thoughts to specific design goals and major features to satisfy the user's main needs and desires for having a code review in the light of creativity support tools and crowd computation. Three major challenges and features that may resolve those issues are:

1. Challenge: Lack of programming capabilities to create digital artifact Design Goal: Give users the opportunity to learn through code review through embedded IDE plugin, maintaining the mental model

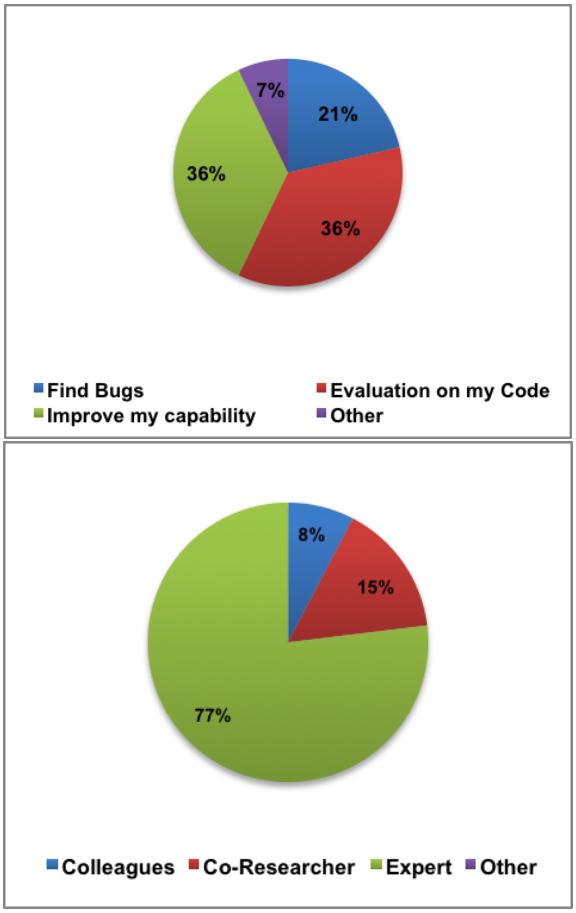

Figure 3. Preferred or appropriate code reviewer (bottom) and reason why they want to have a code review

.

2. Challenge: Lack of knowledge on code review process Design Goal: Give users the opportunity to ask a code review through structured approach

3. Challenge: Lack of opportunity to get code reviewers Design Goal: Connect users to the community that has many domain experts through without contextual changes so that users focus on their work

STACK EXCHANGE CODE REVIEW CHARACTERISTICS

CodeReview (*Code Review Stack Exchange*[7]) is a beta Q&A site for feedback on projects you are working on, by sharing your code with fellow programmers and getting extensive feedback/review of best practices, design pattern usage, application UI, security, performance etc.

CodeReview is quite similar with other StackExchange services; but here is the difference. StackOverflow should be used if you have an actual question about coding. Programmers should be used if you have a conceptual question about coding. CodeReview should be used if you have a bunch of code you want reviewed.

[7] `http://codereview.stackexchange.com`

Even though CodeReview is online just for 2 years, the statistics about the usage of CodeReview are quite nice and these indicate that the community got used to send the code to be reviewed and review the code sent by other developers. Two answers per question are very good in terms of the liveliness of the web service. In overall, CodeReview is in healthy beta mode and is growing day to day. Because it is the best candidate to be used in our crowdsourced code review tool, we implemented the CodeReview interface as a first adaptor to be used in CRODER.

Table 1. Statistics of StackExchange

Questions per day	13.6
The percentage of answered questions	90
Total users	22282
Total users with 200+ reputation	737
Answer ratio	1.9
Number of visits per day	5644

Table 2. Popular Languages in CodeReview

C#	1326
Javascript	1000
Java	906
Php	887
Python	784
C++	680

Table 3. Popular Tags in CodeReview

Best-practice	285
Performance	255
Refactoring	250
Design-patterns	225
Clean-code	189
Readability	148
Security	95

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 4 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 5 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.

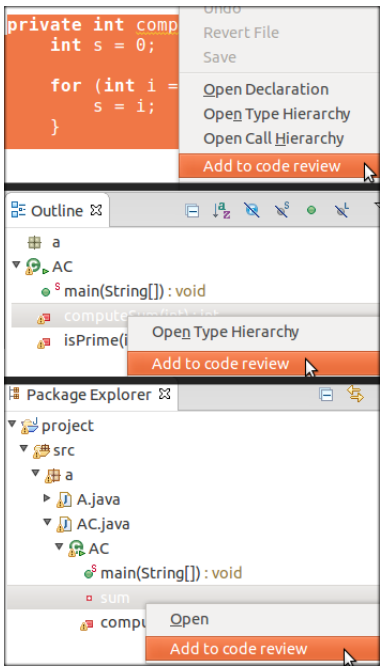

Figure 4. Adding code snippets from various resources

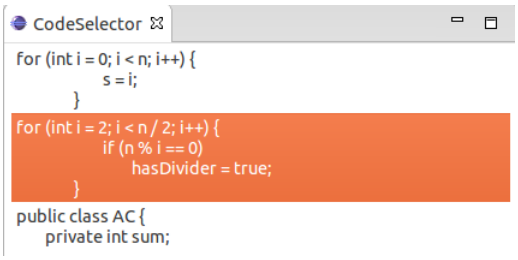

Figure 5. Viewing code snippets

CROWD SOURCED PEER REVIEW CREATION

For creating of the task we chose to implement a wizard. One of the main features is that you can ask for a *structured* code review. The user can select a criteria under which code review should be performed (performance, design, readability etc.). This helps reviewers 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 6 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

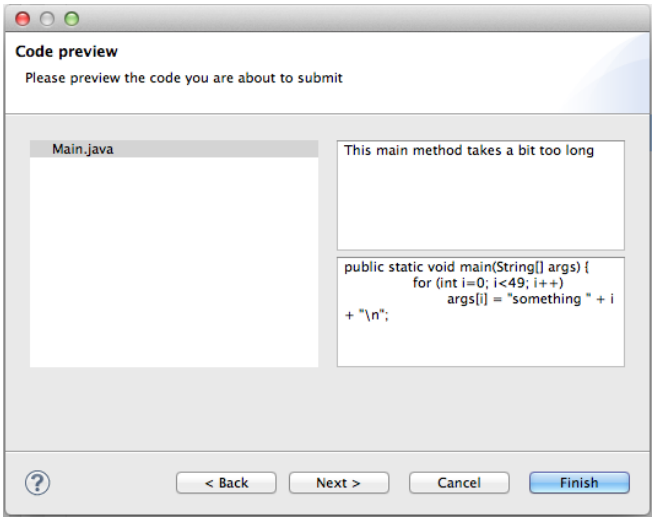

Figure 6. Adding comments for each snippet

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.

```
*/
public static void main(String[] args) {
    for (int i=0; i<49; i++)
        args[i] = "something " + i + "\n";

    args.toString();
```

Figure 7. The market that corresponds to the review

Figure 7 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 9 illustrates the Review Browser. The left pane shows the review titles while the right pane shows the replies for the currently selected review.

Since the code review results may take several days to arrive, and since the programmer may wish to keep certain reviews for future reference, CRODER constantly persists review related artefacts such as snippets, code markers and review replies. Moreover, the persisted files can be moved from one *IDE* installation to another.

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

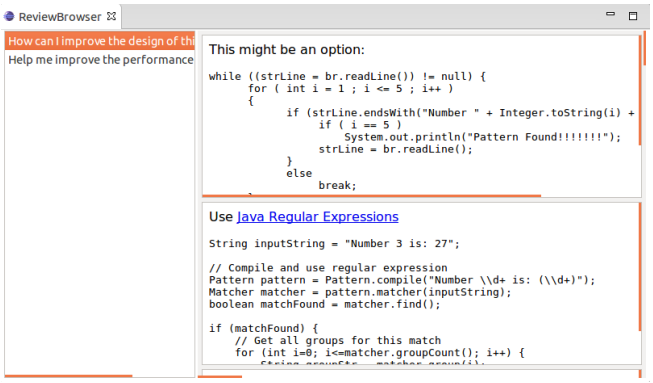

Figure 8. Viewing code snippets

INTERFACE WITH STACK EXCHANGE

Our vision is that CRODER can be used with many services: oDesk, top coder, stackexchange. We implemented the CodeReview (*Code Review Stack Exchange* interface as a first adaptor to be used in CRODER because it is the best candidate among the other services.

Even though getting/posting data from/to a website or a web service seems to be an easy task, this does not apply for the StackExchange network, in particular for CodeReview. There are many challenges to take into consideration when dealing especially with writing access. Moreover, the level of difficulty increases when the data needs to be manipulated and structured.

In the first place, the StackExchange network provides a public RESTful API to interact with its own data. The network does not only comprise of CodeReview, but it consists also of a series of websites (e.g., StackOverflow, gamedev.stackexchange.com), as well as other websites treating completely unrelated topics (e.g., cooking.stackexchange.com and android.stackexchange.com). The data of all of web sites are accessible through the API such that every information regarding users, posts, tags etc. can be retrieved in JSON format. Such a web API can be useful to facilitate the data collection but not to provide write accesses (e.g., posting questions, replying comments).

As the API documentation states, the capabilities are limited on purpose to get write accesses for the service. In v2.1, they introduced write accesses. Unfortunately, the StackExchange policy regarding the writing access is very strict. It requires many reputation points from the member who requests the access and supports only three accesses per day to be performed for writing. Because these limitations can definitely harm the user experience, we eliminated this option completely.

In the second place, instead of the idea of using a public RESTful API, we decided to perform getting/posting data directly on the Stack Exchange website of interest. CodeReview. This would require to implement a webpage scraper to parse the webpages and extract the data from the questions. This solution requires more effort to be implemented than using RESTful API.

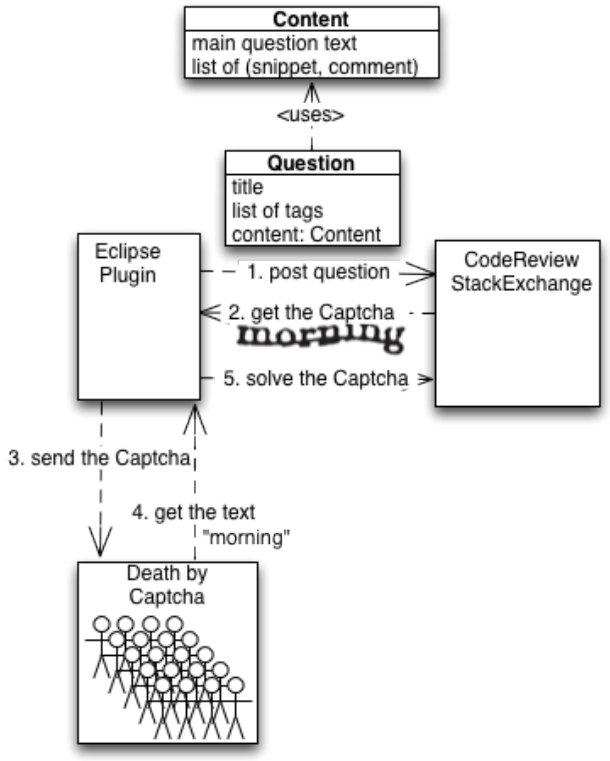

Figure 9. Main architecture of CRODER

Web Scrapping API

We are using HtmlUnit[8] for web scraping. It is a headless web browser written in Java and allows high-level manipulation of websites from other Java code, including filling and submitting forms and clicking hyperlinks. It also provides access to the structure and the details within received web pages. For instance, a sequence such as getPage(url), getLinkWith("Click here"), click() allows a user to navigate through hypertext and obtain web pages that include HTML, JavaScript, Ajax and cookies. Even though it has fairly good JavaScript support, it does not support the latest jquery libraries. Unfortunately, the stackexchange network heavily depends on jquery libraries. We explain the challenges caused by this problem in next sections.

Login

Posting questions requires logging in the service, CodeReview. There are many options for logging in the StackExchange network services: global stackexchange, google, facebook, yahoo, any openid supported accounts. Because google account is the most popular among these accounts, CRODER requires the authentication information of the user's google account. The adaptor API uses this information for filling login form in google. If the account has not created before and it is the first login by using google account, CRODER does not hesitate the developer and initializes the account for CodeReview by auto-clicking some URLs. After login, the adaptor API uses the same web client created by web scrapping API throughout the same eclipse session just because of using cookies for the login.

Post Question

After the user selects the code to be reviewed and added the comments, CRODER wraps these and sends these to the CodeReview adaptor. The adaptor formats this content so that the question is displayed nice in the service. After formatting content and transforming to the text beautified by html code, the interface opens the url of posting question and fills the form (see Figure 10). The main form has three parts required to be completed: title, content, tags. After filling these three fields and submitting the form, CRODER experiences the main challenge. Because jquery is not supported by the web scraping API, javascript is automatically disabled. When javascript is disabled, CodeReview assumes that the visitor is not a human and requires the visitor to solve the popular ReCaptcha[9] challenge (see Figure 11). Because HtmlUnit is not obviously be able to solve the ReCaptcha, the adaptor API sends the ReCaptcha image to the "Death by Captcha"[10] API and gets the text for the corresponding image.

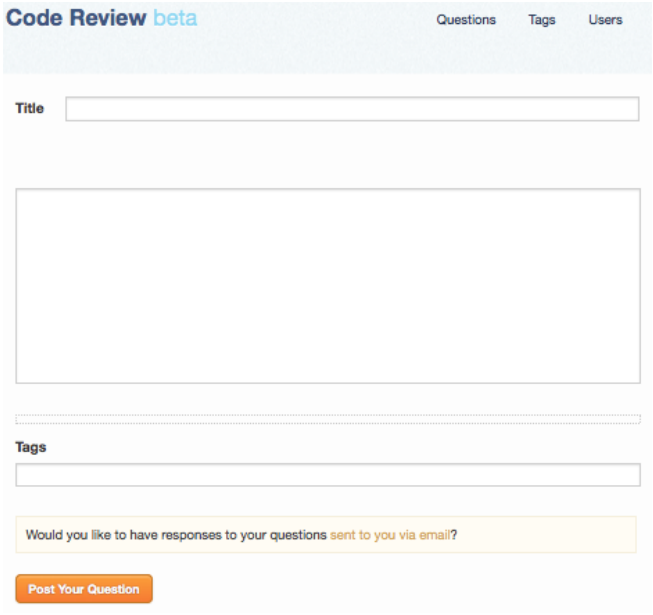

Figure 10. Post Question Form

Crowdsourced Captcha Solver

"Death by Captcha" service aims to solve any Captcha by providing an API. They offer this service from a low price of $1.39 for 1000 solved Captchas. Their solution is a hybrid system composed of the most advanced OCR system on the market, along with a 247 team of Captcha solvers. The average response time is 15 seconds, with an average accuracy rate of >90%. And the customer always pays for correctly solved Captcha only.

[8] `http://htmlunit.sourceforge.net/`

[9] `http://www.google.com/recaptcha`

[10] `http://deathbycaptcha.com/`

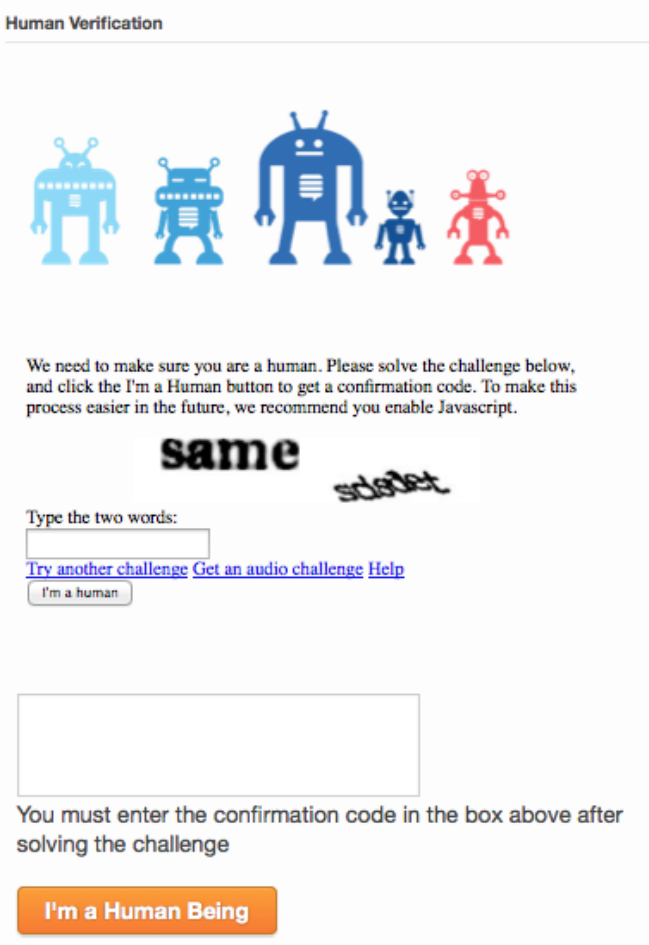

Figure 11. The ReCaptcha Challenge in CodeReview

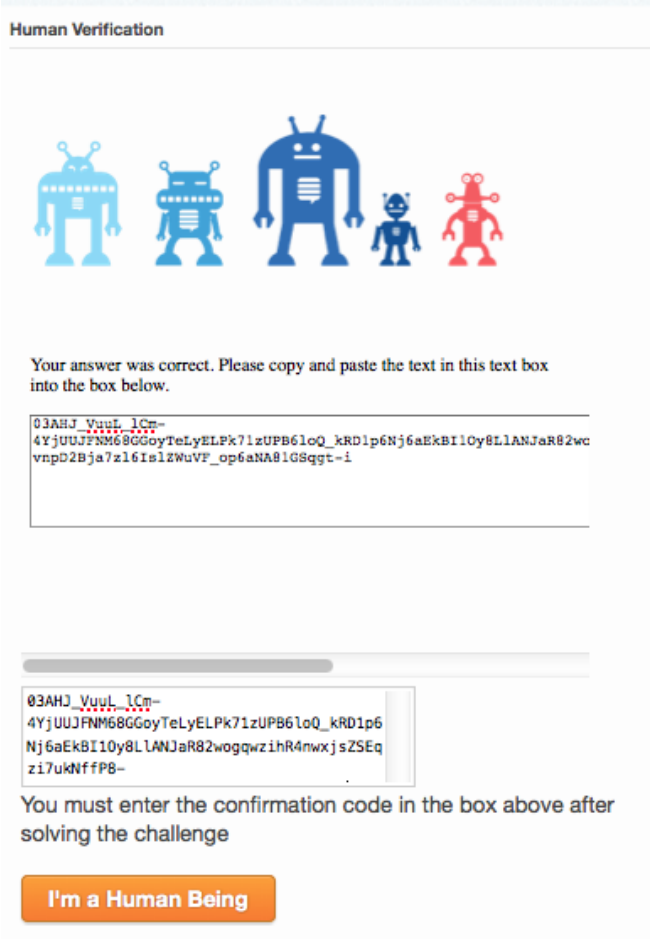

Figure 12. The Confirmation after Solving Captcha

If "Death by Captcha" cannot solve the image at the first try, the adaptor API sends the new image again and again until the client is forwarded to that confirmation page (see Figure 12). In that page, HtmlUnit API copies the confirmation token in the page, pastes it to the that form, and submits the form. Then, the client is successfully forwarded to the URL of the question and stores this URL for checking the replies.

Get Replies

The adaptor API periodically checks whether there is a response to the previously asked questions. If there is a new response, a red colored marker shows up and notifies the developer.

However, as a disadvantage of this solution, it sounds as an unstable solution since the maintenance of such a adaptor API would require a new adaptor API that use the web scraping API whenever the page layout is changed.

SKETCHES

Based on user interview study, we started to draw some initial sketches of the workflow. At the beginning of drawing the sketches, we focused simplifying the interface. Many interviewees mentioned that they have a really hard time beginning to learn code. This is especially true for designers, architects etc. – because they do not have any professional education background in computer science or programming. So, based on the simplification principle, we hide many features in main menu, make it more user-friendly and still have consistency and a hierarchy between different features (see Figure 13).

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 14). Using the metaphor method, the interface is like a hand-made leather cover notebook. This interface can recall the users memory of using notebook to take notes in class. This can make first time users, especially user without background in programming, to quickly become familiar with programming.

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

	Primary User	Secondary User	Tertiary User
Pseudoname	Ben	Kyle	Karla
Education	PhD Student	MFA Graduate	MBA Student
Type	Pragmatic	Opportunistic	Exploratory
Attitude	Writes code methodically. Develops sufficient understanding of a technology to enable component use of it. Prides himself or herself to building robust application or technical challenges.	Writes code in an exploratory fashion. Develops a fundamental understanding of a technology to understand how it can solve a specific problem	Barely write code all by herself. Develop with limited understanding of Java language and she is having a trouble with developing her ideas with a basic syntax quite occasionally.
Motto	I want to focus on creating something new while somebody cleans up my slight mistakes.	I sometimes feel dizzy because I can't find what is wrong with my code.	I barely started to write code in Java to prove my model in the class. But I don't know how to.
Computer literacy	Expert. Majored in CS	Intermediate. Majored in Art and Design	Basic Majored in Business
Pair programming experience	Java, C, C++, Matlab etc.	Adobe ActionScript, HTML/CSS, Proce55ing, Java	Visual Basic, Mathematica
Three most frequent sites they use when they are programming	GitHub, StackOverflow, Codeguru	Google, StackOverflow, Open Processing	Google, eHow, Quora
Goal	Find related info; Find an optimized process or algorithm	Find related info; Find previous Q&A that may help his problem	Find source code; Looking for someone who can answer her questions.

Table 4. Persona table to show their situation, background and goal to code review

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 Crowds for software development, 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 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

We propose a crowdsourced way of doing code reviews. Since code reviews take up a lot of time and require a great amount of coordination, we demonstrate using the crowds to fulfill this task. By integrating the IDE with the review platform we make it easier for developers to review code and have code reviews. Our tool, CRODER can create reviews and get

results from CodeReview StackExchange with hiding all the interactions on the backend. This integration helps developers post tasks without the need of context switching.

REFERENCES

1. Adamczyk, P., and Bailey, B. If not now, when?: the effects of interruption at different moments within task execution. *Proceedings of ACM CHI 2004 Conference on Human Factors in Computing Systems 6*, 1 (2004), 271–278.
2. Capiluppi, A., Lago, P., and Morisio, M. Characteristics of open source projects. In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, IEEE (2003), 317–327.
3. 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.
4. Fink, L. D. Beyond small groups: Harnessing the extraordinary power of learning teams. *Team-based learning: a transformative use of small groups. Praeger Press, Westport, Conn* (2002).
5. Fink, L. D. *Creating significant learning experiences: An integrated approach to designing college courses*. Jossey-Bass, 2003.
6. German, D. Decentralized open source global software development, the gnome experience. *Journal of Software Process: Improvement and Practice 8*, 4 (2004), 201–215.
7. German, D. M. Using software trails to reconstruct the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice 16*, 6 (2004), 367–384.
8. Mamykina, L., Manoim, B., and Mittal, M. Design lessons from the fastest q&a site in the west. *Proceedings of the ...* (2011), 2857.
9. 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.
10. Michaelsen, L. K., Knight, A. B., and Fink, L. D. *Team-based learning: A transformative use of small groups*. Praeger Pub Text, 2002.
11. Mockus, A., Fielding, R. T., and Herbsleb, J. D. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM) 11*, 3 (2002), 309–346.
12. Spolsky, J. Human Task Switches Considered Harmful.

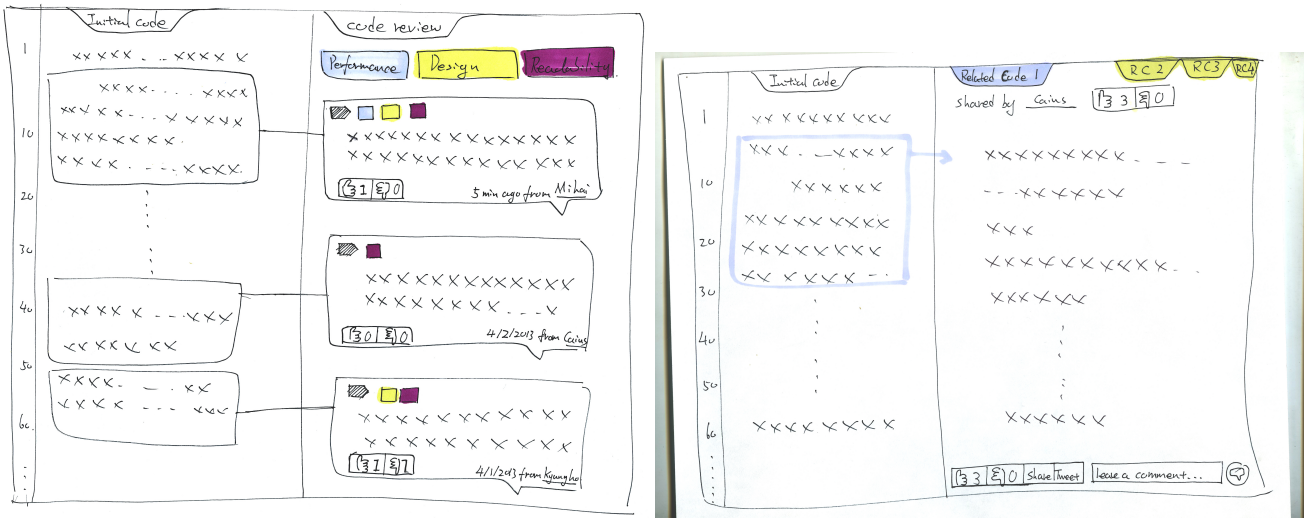

Figure 13. APPENDIX: Initial design sketches

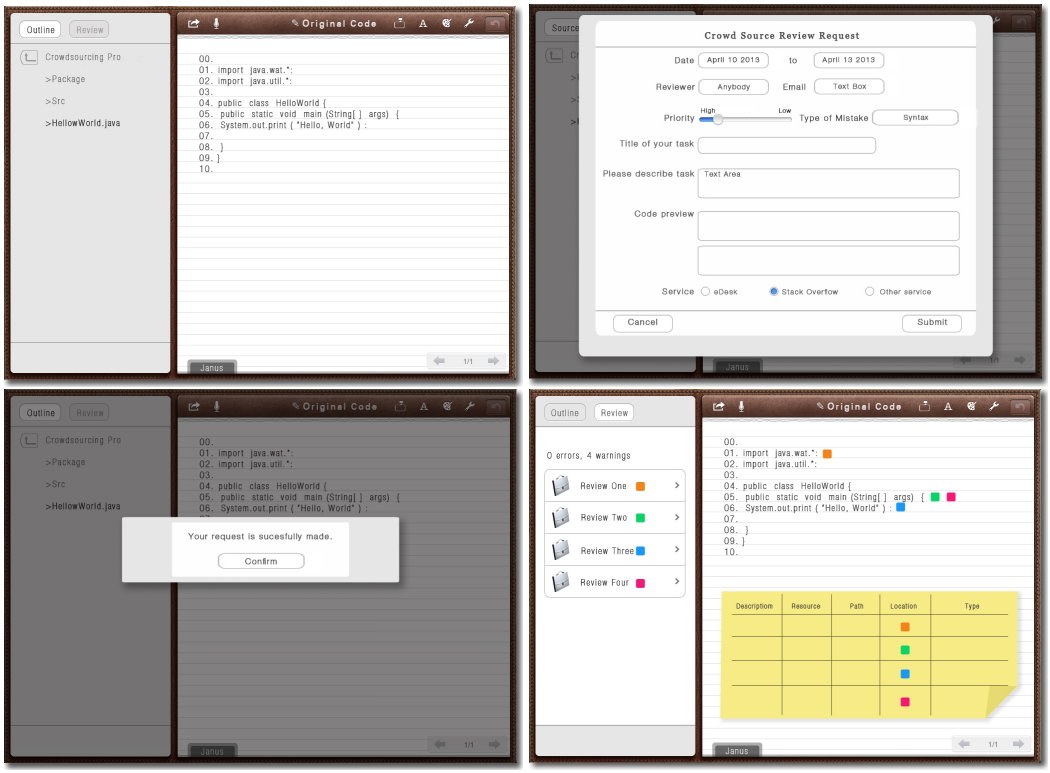

Figure 14. APPENDIX: Interface example