Microtask Programming: Building Software with a Crowd

Thomas D. LaToza[1], W. Ben Towne[2], Christian M. Adriano[1], André van der Hoek[1]
[1]University of California, Irvine
Irvine, CA
{tlatoza, adrianoc, andre}@ics.uci.edu
[2]Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA
wbt@cs.cmu.edu

ABSTRACT
Microtask crowdsourcing organizes complex work into workflows, decomposing large tasks into small, relatively independent microtasks. Applied to software development, this model might increase participation in open source software development by lowering the barriers to contribution and dramatically decrease time to market by increasing the parallelism in development work. To explore this idea, we have developed an approach to decomposing programming work into microtasks. Work is coordinated through tracking changes to a graph of artifacts, generating appropriate microtasks and propagating change notifications to artifacts with dependencies. We have implemented our approach in CrowdCode, a cloud IDE for crowd development. To evaluate the feasibility of microtask programming, we performed a small study and found that a small crowd of 12 workers was able to successfully write 480 lines of code and 61 unit tests in 14.25 person-hours of time.

Author Keywords
crowdsourcing; development environment; programming tools

ACM Classification Keywords
D.2.6 Programming environments: Interactive environments

INTRODUCTION
Microtask crowdsourcing systems enable crowds of workers of varying skill to complete large tasks quickly by decomposing work into short, self-contained microtasks, enabling mass contribution through low barriers to contribution and work to be completed quickly through extreme parallelism. This paradigm has a great potential appeal for software work: while open source development has brought open contribution to software work, joining an open source project is often a long and tedious process, discouraging contribution and reducing the pool of participants. Even in commercial development organizations, there is often a need to build software quickly, as time to market is often valuable. While microtasking may introduce overhead and thereby reduce the efficiency of the development process, there may be situations in which greatly broadening the pool of potential participants can lead work to be completed more quickly through larger scale and parallelization.

Figure 1. An example of a microtask in CrowdCode.

Programming is an example of complex work, involving many interdependencies among components of the work produced. Recent crowdsourcing work has begun to explore approaches for microtasking complex work. For example, CrowdForge [12] introduces a Map-Reduce style paradigm in which the crowd first partitions a large problem into several smaller sub-problems, then solves the sub-problems (map), and finally merges the multiple results to a single result (reduce). However, an important limitation of existing workflows for complex work is that the decomposition structure is *static* and fixed by the requestor. For example, while a requestor might specify a workflow in which workers first partition work into sub-problems before workers then perform a map step, the workflow itself is fixed and cannot vary in response to the work done. For many creative tasks, this is an important limitation. For example, in programming, it is impossible to specify, a priori, the set of functions and tests necessary to implement a program. In

UIST 2014, October 5–8, 2014, Honolulu, HI, USA..

ACM 978-1-4503-3069-5/14/10…$15.00.
http://dx.doi.org/10.1145/2642918.2647349

the process of implementing functions, developers may discover new parts of the problem, requiring new functions to be written. Then, in the process of writing functions, developers may discover they must change their interface, requiring changes to be made to functions elsewhere.

Here, we present an approach for crowdsourcing problems using *dynamically* generated microtasks and illustrate this approach through the design of a system for microtask programming. Our key insight is to coordinate work through a graph of artifacts, generating microtasks in response to events that occur on artifacts rather than through a static workflow. Each microtask asks workers to perform a short well-defined task on a single artifact – a function or a test (e.g., Figure 1), allowing work to proceed on many artifacts in parallel. As workers complete microtasks, events are generated on the artifact, which may then trigger further microtasks to be generated. When an artifact changes, events are sent to artifacts that depend on it, allowing microtask structures to be dynamic and non-hierarchic. For example, when a function changes its signature (e.g., adding a parameter), artifacts that depend on it (callers and tests) are notified, generating microtasks to handle these changes. As artifacts may have many dependencies, artifacts may have multiple pending notifications of changes. To coordinate this work, each artifact has a microtask queue, allowing changes to be performed sequentially and preventing conflicts.

We implemented our approach in a prototype online IDE for microtask programming for Javascript: CrowdCode. Our approach has a number of important limitations: it does not support design tasks, does not crowdsource the design of data types, is limited to crowdsourcing small functional libraries, and requires the correctness of work to be evaluated solely through tests. Within this limited scope, we have explored an approach for microtasking writing code, writing tests, and debugging. To achieve this, we present a novel approach for the dynamic generation of microtasks through an artifact network, a microtask decomposition of programming, and self-contained microtasks for programming. To evaluate the possibility of a small crowd working on programming microtasks in parallel and to evaluate the basic feasibility of the approach, we performed a small user study in which 12 participants worked on a small programming task. We found that the participants were able to successfully program part of a library, completing 265 microtasks, writing 480 lines of code across 16 functions, and an additional 61 unit tests. We found that decontextualizing programming work had both strengths and weaknesses; but, overall, 11 of the 12 participants felt that a microtasking approach would make them more likely to contribute to an open source project.

RELATED WORK

Our research builds on work across several communities: open source software development, crowdsourcing complex work, and crowdsourcing software development. In open source, workers complete tasks to accrue status [4]. Yet this process differs fundamentally from microtasking, as tasks exist at a far larger granularity of hours or days. Workers face many barriers to contributing, including discovering ways to contribute, learning about tools, and tolerating harsh feedback from senior members [10][14][21]. Our approach is intended to reduce these barriers by decomposing work into microtasks, which take only minutes.

Complex work comprises interdependent tasks that require more cognitive effort than the typical tasks of labeling and transcribing data. Approaches to tackle interdependent and complex tasks rely on workflow mechanisms and crowd algorithms. For example, Soylent [2] enables a writer to partition work in smaller proof reading and editing tasks to be performed by a crowd. TurKit [16], provides a framework based on scripts to create and run tasks in Mechanical Turk. CrowdForge [12] expands those solutions by enabling the crowd to partition work. Our approach extends these models, supporting dynamic, non-hierarchic workflows.

Other work has begun to apply microtasking to programming at the level of individual development tasks such as testing or question answering. In Stack Overflow[1], developers ask questions, other developers answer them, and yet other developers evaluate the quality of the answers, concurrently curating a knowledge repository of frequent questions [11,17]. Other work has explored the use of crowdsourcing for recommending fixes to bugs [9,18] and compilation errors [22] and to checking and fixing unit test assertions [19]. In order to leverage larger pools of workers, some systems enable non-specialists to contribute. For instance, several systems have explored applying a gamification paradigm to verifying software models for correctness [15] or verifying for security vulnerabilities by playing with pipes [5].

One of the few systems to explore microtasking a programming process is Collabode [7,8]. In Collabode, an "original programmer" describes in prose short microtasks to be performed and workers then use a provided web IDE to complete the requested tasks. An evaluation of the system found that, while it was possible to microtask programming, there were several significant issues with the workflow used. As workers all worked with a global view of the entire codebase, it was sometimes distracting to see changes being made elsewhere. And there was a large overhead for the requestor in managing the crowd workers, as they needed to answer questions about the request and evaluate the work in detail. Moreover, code often had subtle bugs, which was difficult for the requestor to find through code inspection. Finally, workers were anonymous and thus sometimes did not take responsibility for their work. These considerations directly influenced our design choices in

[1] stackoverflow.com

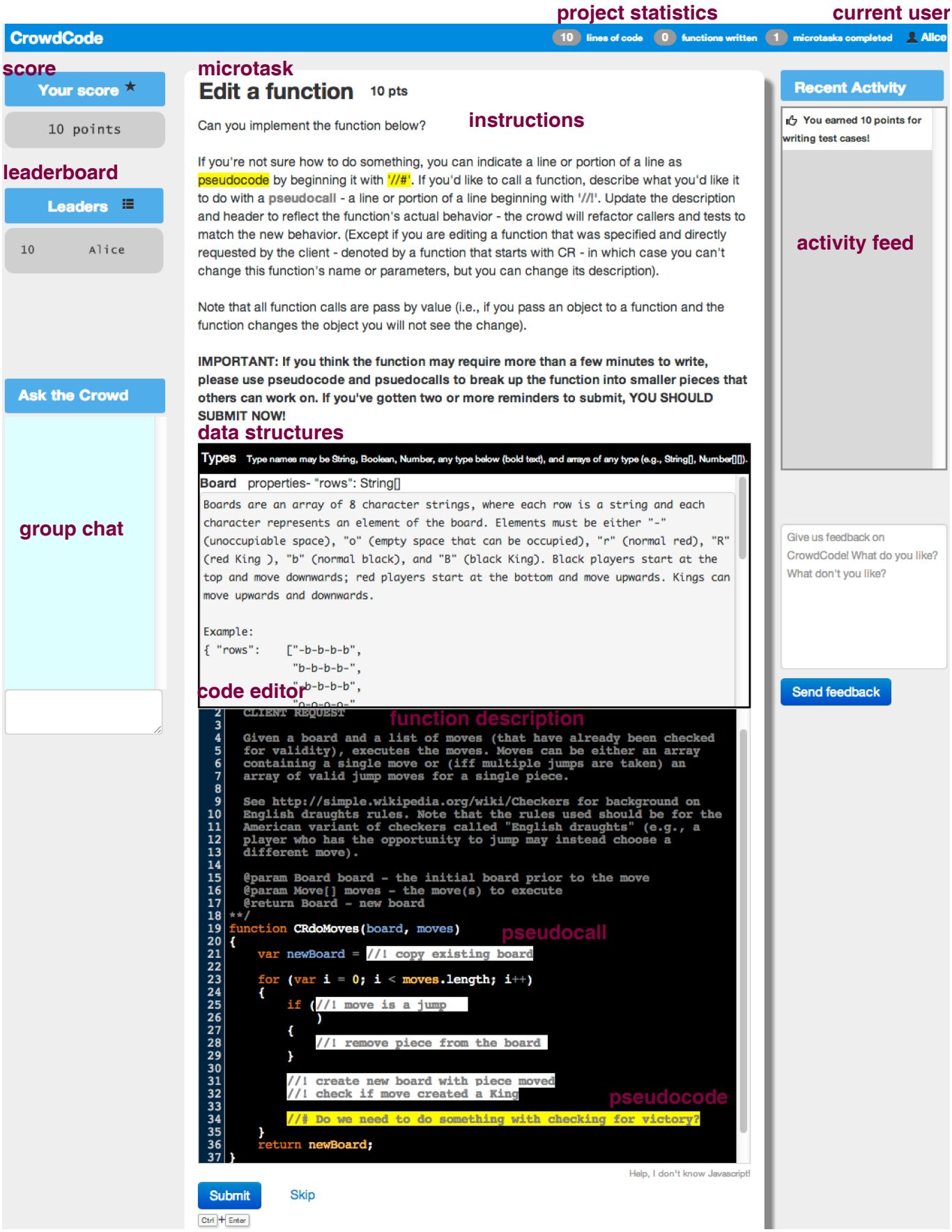

Figure 2. The CrowdCode environment and the *Write Function* microtask.

CrowdCode, leading us to adopt a model with local, self-contained microtasks and test-based correctness evaluation.

EXAMPLE

To illustrate microtask programming in CrowdCode, we present an example. After logging in to CrowdCode and viewing a welcome screen, Alice is immediately presented with her first microtask. The microtask provides her with the description of a function in prose and asks her to enumerate test cases.

Not feeling in the mood for testing, she clicks the skip link at the bottom of the page. She's then presented a new microtask – *Write Function* (Figure 2) – and asked to write some code. Rather than completely implement the function, she sketches some pieces of it that come to mind, noting portions still to be done with pseudocode (yellow background). She thinks some of the functionality should really be implemented in other functions, and writes several pseudocalls describing what she thinks those functions should do. She submits the task.

She next receives a microtask to *Debug*, and is given some code and unit tests and sees that the unit test is failing. She edits the code, but the unit tests still are not passing. Looking at a list of inputs and outputs for function calls at the bottom, she sees that one of the functions is returning an erroneous value. After editing the output value, she reruns the tests, sees that they pass, and submits. She sees that her score has now increased to 20 points!

Alice next is assigned another microtask to edit a function, and sees that she has been reassigned the microtask for the function she started working on earlier. But it has now changed – some of the pseudocode she had written has now been replaced with code, and several of the pseudocalls have been replaced with actual calls. But she also sees that some of the new algorithm does not appear to work correctly, so she rewrites some of the code, adding new pseudocode and pseudocalls for some of the new portions.

DESIGN

The core of CrowdCode is a system for tracking work as a graph of artifacts, dynamically generating microtasks in response to state changes in artifacts and propagating events across dependencies. To enable workers to program using these microtasks, CrowdCode decomposes programming work into a set of microtasks, enabling workers to write code, reuse functions, test, and debug within self-contained microtasks. Finally, CrowdCode provides social features to motivate contributions including a simple point system. The following sections greater detail CrowdCode's design and concepts.

Generating Microtasks

In CrowdCode, all work performed by the crowd occurs in microtasks. A *microtask* is a short, independent, self-contained request for a piece of work to be completed. Each microtask focuses on a single *artifact* – a work product being produced by the crowd. After a crowd worker completes and submits a microtask, the microtask's corresponding artifact processes the work completed, updating its data and state accordingly.

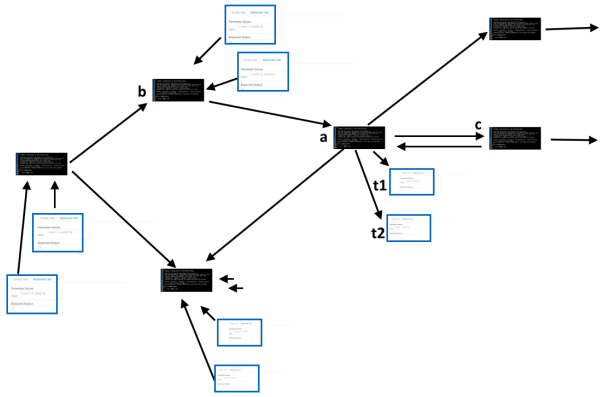

Figure 3. A graph of artifacts with functions (black background), tests (blue outline), and dependencies (arrows).

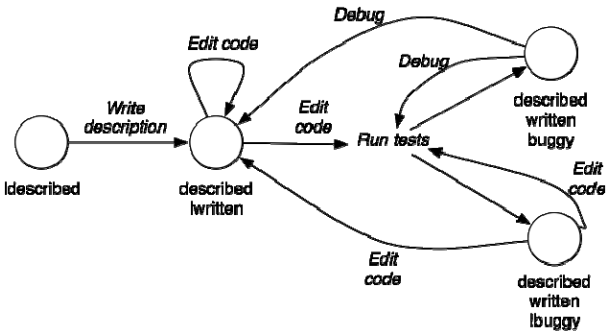

Figure 4. The function state machine.

In CrowdCode, the overall work product is maintained as a graph of artifacts (Figure 3). Each artifact – functions, tests, and the project – includes a set of attributes describing its *state*, describing the work which has been completed and the work which may be required. When a microtask is submitted, an artifact may change state, transitioning attributes as necessary. For example, when the *Write Function Description* microtask is submitted, a function changes state from *not described* to *described.* See Figure 4 for the function state machine.

Microtask submissions may also trigger an artifact to send an *event* to other artifacts that depends on it. For example, when a parameter is added to a function's signature, the function sends a signature change event to all functions that call it and all of the unit tests for the function, enabling these artifacts to generate microtasks in response. In Figure 3, adding a parameter to function *a* results in events being sent to functions *b* and *c* and tests *t1* and *t2*. In this way, changes to an artifact may propagate across the call graph, allowing related artifacts to be updated. CrowdCode cur-

rently implements two types of dependencies: function calls and tests that test a function.

CrowdCode also supports iterative workflows, where microtasks for the same work are repeatedly generated until work has been completed. If, after a microtask is submitted, the artifact's data is updated but it remains in the same state, it may generate a new microtask to continue the work. For example, developers editing a function can write *pseudocode*, leaving the state of the artifact in the *not written* state and iteratively generating microtasks until all of the pseudocode has been replaced with code.

Organizing work through explicit global tasks is challenging and fragile. As workers do work at scale throughout the system, in parallel throughout the system, this work must then be reassembled into a consistent whole. In our early work, we explored the use of *global* tasks, spanning multiple artifacts. For example, a debugging task might spawn a set of microtasks to be done on a set of functions, and would not be completed until the bug was definitively located in a function. However, as each function may be concurrently changing, a single function might concurrently participate in multiple debugging tasks all while other work is being done. Determining if a bug has been found, when the buggy function might have been concurrently changed as part of another task, was challenging.

As a result, we instead adopted a simpler, but powerful principle: each artifact may have a single microtask concurrently being performed and each microtask must act independently. When there are multiple microtasks to be done (e.g., a function fails a test and must also update a call to match a changed signature), each microtask is added to a queue. Each artifact ensures that it has only a single available microtask in the global queue at a time. Other microtasks are maintained in a separate per-artifact queue and released into the global queue as microtasks are completed. This design prevents merge conflicts, as only at most one microtask and worker have commit access to an artifact at any point in time. This poses the secondary issue that abandoned tasks could hold up development on an artifact, and timers that strictly or arbitrarily limit total microtask time might cause issues for e.g. function implementation tasks that a worker is spending more time on. Our implementation uses soft notices to submit after 8 minutes and includes inactivity timers, to help address this issue.

Workflow

CrowdCode crowdsources the implementation of libraries requested by a client developer. All work in CrowdCode begins with a client request specifying the API of a library to be implemented by the crowd. Clients describe an API through a set of functions, each containing a prose description of the functions purpose and its *signature*, including its name, return type, and list of parameters (including name, type, and text description/interpretation). Together, these functions describe the behavior of a library that can be incorporated by the client into a larger codebase. Clients also specify a set of data types, allowing each function to have a type describing the legal values that may be passed to and returned from the client and to be used internally within the library's implementation.

A central decision in the design of a crowdsourcing system is the granularity at which workers interact with the work products produced. A smaller granularity enables greater parallelism, as it increases the amount of crowd workers that can be working at the same time, in turn decreasing the time required to complete work. However, decomposing work into smaller pieces can also increase the amount of overhead, as more workers may need to understand some of the same aspects of the current status of the work to contribute.

CrowdCode attempts to balance these factors towards the smaller end of the granularity scale, using the function as the central unit of granularity. Functions are a natural and central boundary in programs, enabling a set of related statements to be organized into a coherent whole providing a single piece of functionality. Functions are a central unit of abstraction in programs, providing an interface through which clients may invoke the described functionality without seeing or reasoning about the code providing the function's implementation.

As workers work with code in CrowdCode, workers interact with a single function at a time. Through the function's description, workers can understand what callers expect of the function, enabling them to reason about and work on the function in isolation from the code of the other functions. Similarly, workers may request some functionality to be created (or reused) in the system. This request leads the crowd to find or create a function, which can then be called from the requesting function. In this way, functions and their interfaces provide boundaries establishing individual units of work that can each evolve separately.

Another central consideration in a crowdsourcing system is providing mechanisms to produce quality work. In CrowdCode, this is done through unit tests. Drawing inspiration from test-driven development [1], microtasks are separately created to write tests for each function. When all of the tests pass, the system is considered to be ready for acceptance testing by the client. If a test fails, a microtask is created to debug the function. By creating separate microtasks for writing code and testing, CrowdCode creates redundancy, ensuring that the code produced is correct enough that it passes its tests. If workers produce code that fails to pass the tests, more work will be created until the tests pass. Of course, the tests themselves may also be incorrect. When debugging, workers may report an issue with a test, generating a microtask to address the issue and correct the test.

As a result of requiring that bugs be able to be detected through tests, CrowdCode requires that code written is functional and neither mutates global state nor interacts

Microtask type	Description	Possible subsequent microtasks
Write Function	Sketch or implement a function using code, pseudocode, and pseudocalls.	*Write Function, Reuse Search, Machine Unit Test*
Reuse Search	Given a pseudocall and the surrounding code, determine if an existing function provides the functionality or that no function does.	*Write Call, Write Function Description*
Write Function Description	Given a pseudocall and the surrounding code, write a description and signature for a new function for this behavior.	*Write Call*
Write Call	Replace the specified pseudocall with a call to the specified function or edit the function to implement the behavior in an alternative way.	*Write Function, Reuse Search, Machine Unit Test*
Write Test Cases	Given a description of a function, list test cases.	*Write Test*
Write Test	Given a test case and the description of a function, implement the test case or report an issue in the test case.	*Machine Unit Test, Write Test Cases*
Machine Unit Test	Executes all implemented tests, notifying functions if they fail a test	*Debug*
Debug	Edit code to fix bug, report an issue in a test, or create stubs describing issues in function call	*Machine Unit Test, Write Function, Reuse Search*

Table 1. The microtasks in CrowdCode.

with the external environment (e.g., writing output to a screen). Functions must be able to be completely specified simply by evaluating the output they produce for each set of inputs. This enables functions to be evaluated for correctness simply by seeing if, for all tests, they produce the correct output. While this does not allow CrowdCode to write web apps with GUIs, CrowdCode can be used to write a library implementing key behavior as part of a larger application containing a GUI and other interactions with an environment.

Table 1 lists the microtasks in CrowdCode, which are discussed in detail in the following sections.

Writing code

Writing code involves a number of distinct tasks – writing descriptions for functions, envisioning and sketching a high-level outline or algorithm, implementing the sketch with code, locating existing functions to reuse or describing new functions to be created, adding function calls, and responding to changes in the interface of functions being called. In CrowdCode, each of these are separate microtasks performed by the crowd.

The first functions are initially requested and described by the client. The crowd begins contributing through a *Write Function* microtask (Figure 2). Workers are provided a description of the function and its signature and asked to begin implementing it. As workers begin writing the function, they may choose to simply sketch portions of the code using *pseudocode*. Workers may indicate that a portion of a line is pseudocode by the leading characters //#; pseudocode is indicated visually with a yellow background. Workers may submit incomplete functions with pseudocode, generating additional microtasks to iteratively continue the work [16].

Workers editing a function may also wish to reuse existing functionality or break up the work to be implemented into multiple functions. In CrowdCode, workers do not need to choose between these cases. Workers may write *pseudocalls,* indicated by a portion of a line beginning with //! and visually highlighted with a white background (against the black code editor background), to request that the crowd either locate an existing function with the specified behavior or to create a new function if no such function exists. This allows the worker editing the function to be oblivious to the other functions that may or may not currently exist – they simply request a function, and the crowd determines the most appropriate way to provide it.

CrowdCode also provides error checking. When a function has pseudocalls or pseudocode remaining, error checking of the function's body is suppressed. This allows workers to incorporate pseudocalls and pseudocode into lines of code in ways that makes the code itself syntactically invalid (e.g., branches and loops with some pseudocode components). In early pilot testing, we found that workers often wished to produce such code, and forcing them to always create syntactically valid code was a significant barrier. Whenever a function has no pseudocalls or pseudocode present, CrowdCode provides error checking, displaying an error panel below the code whenever code errors are present. CrowdCode provides basic syntax checking using JSHint[2].

In CrowdCode, workers can only create functions through the pseudocall mechanism. This prevents a single worker from writing a whole program in a single microtask and encourages workers to break the work to be done into additional microtasks, increasing the parallelism of the work

[2] www.jshint.com

process. To enforce this rule, CrowdCode displays an error message whenever the code editor contains more than one function.

Data types serve an important role in communicating the expected parameters of a function, signaling, for example, that the parameter *player* is expected to be a String in our example task described below. Defining good data types is often a central task of high-level design, requiring a global understanding of a code base. In a microtasking environment where no single worker has such a global view, this is challenging to achieve. Moreover, if a crowd were to iteratively create data types, every time the data types changed, all functions and tests with parameters using those data types might need to change, creating the potential for large amounts of work. Thus, in CrowdCode, all data types are defined by the client as part of the initial client request.

Clients specify data types with a name, list of fields, list of data types for each field, and a prose description (e.g., Board in Figure 2). CrowdCode supports nested data types and arrays of data types. Each parameter to a function and the return value must have a type, which is either a data structure or a primitive type (i.e., Number, Boolean, or String). Parameter types are specified in the comments of the function description (e.g., lines 15 – 17 in Figure 2). CrowdCode displays an error message when a provided type name is invalid. Descriptions of all data types in the system are listed above the code editor.

Reuse and creating functions

Whenever a worker submits code with a new pseudocall, a *Reuse Search* microtask is created. This microtask provides the text from the pseudocall and the code surrounding the pseudocall and asks the worker to search through existing function descriptions to determine if such a function already exists. When search text is entered, it is matched against existing descriptions, and a list of matches displayed. Workers can either select one of the functions or indicate that no existing function provides the requested functionality.

When a new function is required, a *Write Function Description* microtask is next generated (Figure 5). This again provides the pseudocall and the code surrounding the pseudocall and provides a structured editor for writing function descriptions. For each parameter, a textbox is provided for workers to provide the name, type, and description. Error checking is performed, checking for syntax errors, ensuring the function name is unique, and ensuring that the types provided are valid types.

After an existing function has been located or a new function has been described in *Write Function Description*, an *Add Call* microtask is generated (Figure 6). Workers are provided a code editor, functionally equivalent to the Edit a Function microtask, but more specific instructions to replace a specified pseudocall (also highlighted in the code) with a call to the described function or determine another

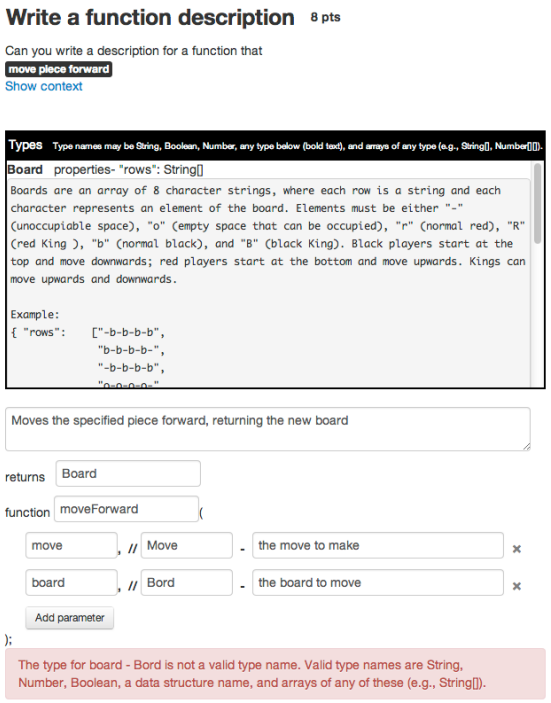

Figure 5. The *Write Function Description* microtask.

Figure 6. The *Add Call* microtask.

way to implement the specified behavior. Workers are free to edit whatever aspects of the code they wish, enabling them to make arbitrary changes in response to the new function or even to decide that a different way of implementing the requested behavior would be more effective. As in the *Write Function* microtask, adding pseudocode or pseudocalls generates the appropriate new microtasks. Whenever a function call is added, a dependency is created on the function by the function being called. As CrowdCode only permits direct calls to functions in the global scope (e.g., calls to functions on objects are not permitted), function calls can always be uniquely resolved to a single function, eliminating the possibility of any false positives or false negatives in creating dependencies.

When working in a function, workers may also decide to edit a function's description or signature. Workers may rename a function; add, remove, or rename parameters; and change the type of any parameter. Any of these changes signals a change in the function's interface. As a result, callers or tests of the function may need to change to reflect the function's new interface. Thus, CrowdCode generates microtasks signaling the description has changed for each caller and test. Each microtask includes a text-based diff of the old and new description and signature, describing the change to the function and allowing the worker to perform an appropriate edit, if necessary.

Testing

Tests are written in two-steps. As soon as a function has been described by a client or by the crowd, a microtask is generated to *Write Test Cases* (Figure 1). Workers are provided a description of the function and asked to enumerate short prose descriptions of test cases. Allowing a single worker to write all of the test cases helps ensure that test cases are not duplicative and have good coverage. To keep the microtask short, workers are asked to provide a prose description of test cases rather than a full implementation.

In the second step, each submitted test case generates a *Write Test* microtask. A worker is provided the function description and test case and asked to concretely specify the test case as a unit test. To make unit tests quicker and easier to write, CrowdCode provides an editor for simple unit tests, asking workers to specify appropriate values for each parameter and the return value (Figure 7). Test values are checked for syntax errors and that they are of the correct type.

If a worker feels that the prose description of a test case is incorrect for the function (e.g. testing an invalid input when the parameter is specified to have been validated), they may report an issue in the test case. This generates a new Write test cases microtask that prompts a different worker to consider the issue and edit, add, and remove the test cases to address the issue. Any changes to a test a case generate a new Write test microtask reporting the change to the test case and asking the worker to edit the test.

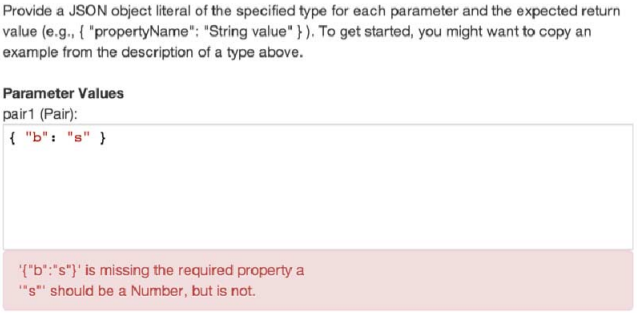

Figure 7. When implementing a unit test, workers are asked to write JSON literals for each parameter, which are checked for syntax and semantics errors.

Determining when tests should be run presents a potential need for global coordination. Generally, tests should be run whenever a function no longer contains pseudocode or pseudocalls (is written) and all of the functions it directly or indirectly calls are written. Global coordination such as this is again fragile: if a microtask is scheduled to run a test for a function and one of the functions it calls concurrently transitions to not written (e.g. by the addition of pseudocode in an editing task responding to a callee signature change), running the tests is no longer required. To prevent this need for global coordination, CrowdCode uses a simpler, local rule. Whenever (1) a function is edited which no longer contains any pseudocalls or pseudocode and (2) all of the functions' tests are currently implemented, the function notifies the project that it is ready to be tested. The project then generates a special *Machine Unit Test* microtask. This microtask requires no work by the worker; the worker simply briefly sees a microtask appear and a progress notification. The machine unit tests executes all implemented tests for all described functions, regardless of if they are written. The body of functions that are not yet written is replaced with an empty body that simply throws a *Not Implemented* exception. When running tests, if a *Not Implemented* exception is encountered, the test result is ignored. Otherwise, if a function fails its tests, the function is notified, transitions to the buggy state, and generates a Debug microtask.

Debugging

Whenever a function fails to pass a unit test, it transitions to the *buggy* state (Figure 4), and a *Debug* microtask is generated. Workers are provided a code editor and a list of unit tests, with passing unit tests listed in green and failing unit tests listed in red. To fix the bug, workers can edit the code, rerun the unit tests, and view the output. A worker may also decide that the issue is not in the code but in the test itself and instead submit a prose description of an issue for the unit test, generating a microtask to edit the test to address the issue.

In other cases, however, the bug may not be in the function under test but in one of the functions it calls. Indeed, much of the challenge of debugging often rests in the process of

fault localization and determining the location within the program where the problem is located. Such a task is non-modular in that it requires developers to navigate the whole program, traversing function calls to determine the location of a fault.

How can workers debug such bugs through local microtasks which provide a view of a single function? Our solution is to allow workers to *edit* the return value of function calls, creating a *stub* overriding the function's return value for a specific set of inputs. For example, a worker might see that the call to the function add with the parameters -1 and 2 is returning -1 and edit the return value to be 1. Workers may then rerun the tests to determine if changing the callee's behavior fixes the bug, with the system automatically substituting the stubs for calls to the actual function through source rewriting. After the microtask is submitted, each stub then generates a test for the callee, which will be run and fail (assuming the callee has not been concurrently changed). A new worker can then continue debugging in the function being called.

Social features

To encourage workers to contribute, CrowdCode implements a simple point system. All microtasks are initially assigned a point value based on their type, approximately proportional to the anticipated difficulty of the task type. Each worker has a score and is awarded the microtask's points when the microtask is submitted. Workers can see the score of all workers in the system on a leaderboard (Figure 2), which is updated in real-time.

When a worker logs in to the system, they are automatically assigned a microtask by the system. Compared to manual task assignment in which workers themselves select microtasks, automatic task assignment has two key advantages. First, workers do not spend time searching for microtasks, increasing the time in which they can be working. Second, by using a queue to assign work to workers, the system can ensure that no microtasks *starve* because workers do not wish to attend to them, even initially. However, automatic task assignment reduces worker motivation, as workers no longer have a choice of work [13]. In order to provide the benefits of both automatic task assignment and choice, CrowdCode lets workers skip microtasks. Skipping a microtask adds the microtask back to the global queue, enabling it to be assigned to the next worker seeking work. To encourage workers to do microtasks that may be undesirable, skipping a microtask increases the points that will be awarded on successful completion of that microtask.

CrowdCode provides a number of features to help workers maintain awareness of the current state of the project. As workers complete microtasks, they are added to a personal activity feed (right side of Figure 2), letting workers track their work. Statistics on the current status of the project – the total lines of code, functions fully written, and microtasks completed (top of Figure 2) – let workers see a summary of overall progress in real-time.

In some cases, workers may require information that is not provided by the current microtask. In these cases, workers may choose to use a group chat with all currently logged in workers, a feature we termed *Ask the Crowd.* While global group chat is ultimately unscalable, we introduced the Ask the Crowd feature as a fail-safe measure to enable the crowd to still make progress in the face of unexpected information needs. It also enables workers to go off topic and forge closer relationships with other workers [13].

CrowdCode ultimately depends on workers in the system to work in ways that produce work for other workers to do, especially through writing code containing pseudocode and pseudocalls. In our early testing, we found that workers sometimes attempted to implement large portions of functionality in a single function rather than using pseudocalls to break the work up into separate functions. To address this issue, workers are explicitly encouraged to use pseudocalls and explicitly prompted after every 8 minutes of work on a microtask to submit, even if their work is incomplete, to create microtasks for other workers to do.

Implementation

CrowdCode is implemented as a web application on Google App Engine[3], providing an infrastructure for seamless technical scaling. All artifact and microtask state is stored server-side in AppEngine. When a worker logs into the system, the browser requests a microtask, transferring the necessary state to the browser. When a worker submits a microtask, the modified state is returned to the server and the state updated. All other information – points, the activity feed, leaderboard, chat – is synchronized across browsers in real-time using Firebase[4].

CrowdCode provides a project model. For each new client request, a project is created with its own artifacts, microtasks, and user statistics. Each project is associated with a unique URL, enabling workers to select a project by visiting its URL.

CrowdCode enables workers to write code in Javascript. This has several advantages. Javascript is currently a popular language whose syntax is well-known, making it more likely that workers can contribute without needing to learn a new language, and making it more likely they will easily be able to find answers to syntax questions on the Web. Moreover, Javascript can be executed client-side, enabling the unit tests to be run in the browser and quickly provide feedback for the debugging microtasks. However, a test that runs in the browser might also execute an infinite loop, causing the browser to hang and the worker to be unable to

[3] developers.google.com/appengine

[4] www.firebase.com

continue. To address this issue, all worker written code is executed on a separate thread using the HTML5 web worker API. Long running tests timeout and fail. The code editor is implemented using the CodeMirror editor[5] and the Esprima ECMAScript parsing infrastructure[6].

USER STUDY

To examine the possibility of a small crowd working on programming microtasks in parallel and to evaluate the basic feasibility of the approach, we performed a small user study. We used email distribution lists and personal contacts to recruit 12 participants from our university, all of whom had and/or were working on graduate degrees in computer science and/or related fields. All participants had prior experience programming in Javascript (average 0.6 years) and at least 6 months of experience in industry as a software developer (average 1.8 years). 11 participants were male and 1 female (P7). Participants were paid $60 for two hours of their time.

All participants participated in a single simultaneous session and were each given their own room to ensure that they were only able to communicate through CrowdCode. Participants were first provided a hands-on tutorial with the system and assigned to separate projects in which they each completed several representative microtasks for 10 – 20 minutes. After completing the tutorial, participants then entered a single project and worked on the primary task. Participants were asked to crowdsource game logic for checkers (i.e., English draughts). The experimenters seeded the project with a client request describing two functions to be written and several simple data types specific to checkers. Throughout the study, two experimenters circulated through participants' rooms and verbally answered questions about how to use CrowdCode (which we recognize as of limited scalability) but did not answer any questions about the work itself. Several participants were international students unfamiliar with the rules of Checkers and made use of the link to the rules we provided.

Midway through the Checkers task, participants were asked to complete a short mid-task survey, asking questions about their experiences and challenges. Fifteen minutes before the conclusion of the study session, all participants were stopped and asked to complete a more extensive post-task survey, containing items about their experiences and perceptions with working in CrowdCode.

Results

The twelve participants each worked for about 1.25 hours in CrowdCode (totaling exactly 14.25 person-hours). In total, participants completed 265 microtasks, wrote 480 lines of code across 16 functions, and an additional 61 unit tests (Table 2). Participants did not finish implementing checkers in the course of the study session.

Microtask Type	Completed	Skipped	Mean completion time (minutes)
Debug	28	2	2.67
Machine Unit Test	16	0	0.17
Reuse Search	30	0	1.84
Add Call	8	1	3.81
Write Function	39	10	5.41
Write Test	99	25	2.84
Write Test Cases	36	7	1.85
Write Function Description	20	3	3.06

Table 2. Microtasks completed and skipped by participants.

One central characteristic of microtasking is a reduced context, enabling microtasks to be self-contained and independent. Participants differed in their reaction to this loss of context. Some found it to be freeing: "I had to keep less context in my head when writing functions, because I could not make assumptions [about] the rest of the program" (P6). Others found it burdensome and wanted other information about the current state of the system that the microtasks did not provide. One participant (P9) also reported that the mental context switching required by microtasks was a hindrance to usability.

A majority of participants agreed that the opportunity for communication beyond what was provided would help them to work more effectively. Participants cited a desire to share technical experience, clarify tasks, ask questions about material that others had written. This may partially reflect the patterns of work to which participants were accustomed. One participant stated that additional communication "might lead to conflicts in the case of disagreements. I thought guiding communication via the work and tasks itself was fairly productive" (P1). Participants used the global chat to socialize and clarify the rules of checkers.

Participants appreciated the ability to specialize in tasks they wanted to do and the ability to contribute according to their knowledge and abilities: "I think that CrowdCode would make me more likely to contribute as I could solve the tasks which I could do, and skip the others. I could take on tasks with higher difficulty as and when I feel comfortable. Hence, CrowdCode would be ideal in working in an open source project… [What I liked best was] collaborative coding - each person can effectively contribute according to his knowledge. For example, a testing person might con-

[5] codemirror.net

[6] esprima.org

tribute for test cases, and skip the code development parts if he feels so" (P11). P1 also reported that "I was willing to be imperfect with my work. It was more important for me to constantly push out new work." This suggests that the iteration process may have created an important "failure for free" condition ([20], Ch. 10) in which the cost of trying something and doing it is less than the cost of figuring out if it's OK to try. Participants found the social features of CrowdCode, especially the points and leaderboard, to be motivating and to "help building a productive vibe to coding" (P10).

11 of the 12 participants agreed that they would be more likely to contribute to an open source project using CrowdCode than with a traditional development process. Each cited the lower barrier to entry and ease of jumping in as opposed to the "taxing" "learning and involvement curve" (P7) of open source projects now, as well as the ability to specialize by skipping some tasks. P1 pointed out that the microtasks could be too constraining for seasoned developers but may be better for someone starting out and understanding a new system.

Work submitted with errors sometimes created issues in the microtasks that derived from the completed work. For example, workers sometimes entered incorrect parameter types in the *Write Function Description* microtask, such as indicating the type of a parameter or return value to be a String when it should be a client-defined data type. As a result, participants in the *Write Test* microtask were forced to write tests with the wrong data types, as they were unable to request a change in the function description.

The study also revealed several usability and platform robustness issues. Participants submitted syntactically invalid code that was not correctly flagged by the system. As a result, some participants were unable to successfully complete the *Debugging* microtask, as the test running infrastructure threw an exception and could not display the results of running the unit tests. Early in the session, some workers were forced to wait to receive microtasks. All of the microtasks were initially spawned in response to a client request for two functions, generating initial microtasks to write the functions and write test cases. Workers writing the functions initially spent a long time working, causing a delay in creating other microtasks for workers to perform.

DISCUSSION

We found that the workers in our study were successfully able to write code and tests within a dynamic microtasking workflow. Especially after the tutorial and early experience with the system, participants seemed to "get it" and found aspects of the system and microtask style that they enjoyed. We were surprised at the motivational power of the points system and leaderboard, especially because participants were well-paid and did not expect the points to have value after the conclusion of the study.

CrowdCode enables developers in a larger project to specify the behavior of a requested functionality as a library through an API (e.g., the API for executing checkers moves in our study), which can be implemented through CrowdCode and added to the project. However, this model imposes a significant burden on the requesting developer: they must precisely specify the behavior of the library, listing descriptions of functions and all necessary data structures. This model might be relaxed by enabling iterative communication between the crowd and requestors, allowing the API to evolve through the joint work of the requesting developer and crowd. Or, in some cases, it may be advantageous to allow requesting developers to play a larger role in the work itself, enabling them to see and direct a global view of the crowd's work.

Microtasking workflows clearly impose an overhead on a development process, and the total amount written per person-hour is likely lower with CrowdCode than with traditional approaches. However, if microtasking is able to successfully reduce the barriers to contribution and thereby harness value from the "long tail" of participation – the many willing to donate small amounts of time and effort – the benefits of tapping into a much larger available resource may still outweigh the overhead costs of using that "free" resource less efficiently.

Another important question is whether or not participants engaged in microtasked work feel that they are making an important and meaningful contribution. On the one hand, microtasked work decontextualizes contributions, which may make it more challenging for workers to understand the impact and significance of their work. On the other hand, by making work products more fine-grained and explicit, it may be possible to provide more information about the impact of work done. For example, a worker writing a function description might receive a notification in the newsfeed whenever the function is reused, letting them see how successfully they were able to craft a reusable API.

A fundamental challenge in crowdsourcing is that workers may produce bad work, even through good faith efforts. In CrowdCode, any of the information workers enter in microtasks may ultimately be wrong and need to be corrected. Unlike more traditional microtasking workflows in which redundant work or explicit reviews are used to ensure the quality of the work [6], CrowdCode embeds corrections into the workflow itself. CrowdCode provides two mechanisms to enable such corrections: workers may directly edit the artifact corresponding to the current microtask or may report an issue with related artifacts that are visible but not editable (e.g., a test case description in Write Test). Workers were often faced with a microtask resulting from bad work. The one area in which this was impossible – reporting an issue with a function signature when writing a test – caused significant problems. This highlights the importance of ensuring that all worker-produced data can be corrected.

Creative work done by large groups often has the structure of separate artifacts with dependencies, leading to corresponding challenges communicating about these dependencies (i.e., socio-technical congruence [3]). The general principles of our approach may apply to many of these domains. For example, in an engineering task, subcomponents may be spun off like pseudocalls, and automated test cases could include static and thermal analyses. In writing text, editing a paragraph in an article might be a microtask, enabling workers to create bullet points fleshed out by the crowd, requests for other related paragraphs to be written, and automatic tracking of dependencies to create microtasks to update work. Our approach may be most useful in contexts where parallelism-based speedups or broad participation through low barriers to entry are needed.

ACKNOWLEDGEMENTS

We thank Steven Morad, Patrick Nguyen, and Eric Chiquillo for their contributions to CrowdCode, we thank the participants in the study for their participation, and we thank Christoph Hannebauer and the anonymous reviewers for their helpful comments and suggestions on previous drafts. This work was supported in part by the National Science Foundation under grants NSF IIS-1111446, IIS-1302522, and CCF-1414197.

REFERENCES

1. Beck, K. *Test-Driven Development: By Example*. Addison-Wesley, Boston, 2003.
2. Bernstein, M.S., Little, G., Miller, R.C., et al. Soylent: A Word Processor with a Crowd Inside. *Proc. of UIST 2010*, 313–322.
3. Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., and Carley, K. M. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. *Proc. of CSCW 2006*, 353-362
4. Crowston, K., Wei, K., Howison, J., and Wiggins, A. Free/Libre Open-source Software Development: What We Know and What We Do Not Know. *ACM Comput. Surv.* 44, 2 (2012), 7:1–7:35.
5. Dietl, W., Dietzel, S., Ernst, M.D., et al. Verification Games: Making Verification Fun. *Proc. of FTfJP 2012*, 42–49.
6. Doan, A., Ramakrishnan, R., and Halevy, A. Y. Crowdsourcing Systems on the World-Wide Web. *Commun. of ACM* 54, 4 (2011), 86-96.
7. Goldman, M., Little, G., and Miller, R.C. Real-time Collaborative Coding in a Web IDE. *Proc. of UIST 2011*, 155–164.
8. Goldman, Max. *Software Development with Real-Time Collaborative Editing*. PhD Diss. Massachusetts Institute of Technology, 2012.
9. Hartmann, B., MacDougall, D., Brandt, J., and Klemmer, S.R. What Would Other Programmers Do: Suggesting Solutions to Error Messages. *Proc. of CHI 2010*, 1019–1028.
10. Jergensen, C., Sarma, A., and Wagstrom, P. The Onion Patch: Migration in Open Source Ecosystems. *Proc. of ESEC/FSE 2011*, 70–80.
11. Jiau, H.C. and Yang, F.-P. Facing Up to the Inequality of Crowdsourced API Documentation. *SIGSOFT Softw. Eng. Notes* 37, 1 (2012), 1–9.
12. Kittur, A., Smus, B., Khamkar, S., and Kraut, R.E. CrowdForge: Crowdsourcing Complex Work. *Proc. of UIST 2011*, 43–52.
13. Kraut, R.E. and Resnick, P. *Building Successful Online Communities: Evidence-Based Social Design*. MIT Press, 2012.
14. Krogh, G. v., Spaeth, S., and Lakhani, K. R. Community, Joining, and Specialization in Open Source Software Innovation: A Case Study. *Research Policy* 32, 7 (2003), 1217–1241.
15. Li, W., Seshia, S.A., and Jha, S. CrowdMine: Towards Crowdsourced Human-Assisted Verification. *Proc. of DAC 2012*, 1254–1255.
16. Little, G., Chilton, L.B., Goldman, M., and Miller, R.C. TurKit: Human Computation Algorithms on Mechanical Turk. *Proc of UIST 2010*, 57–66.
17. Mamykina, L., Manoim, B., Mittal, M., Hripcsak, G., and Hartmann, B. Design Lessons from the Fastest Q&A Site in the West. *Proc. of CHI 2011*, 2857–2866.
18. Mujumdar, D., Kallenbach, M., Liu, B., and Hartmann, B. Crowdsourcing Suggestions to Programming Problems for Dynamic Web Development Languages. *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, ACM (2011), 1525–1530.
19. Pastore, F., Mariani, L., and Fraser, G. CrowdOracles: Can the Crowd Solve the Oracle Problem? *Proc. of ICST 2013*, 342–351.
20. Shirky, C. *Here Comes Everybody: the Power of Organizing Without Organizations*. Penguin, 2008.
21. Steinmacher, I., Silva, M. A. G., and Gerosa, M. A. Barriers Faced by Newcomers to Open Source Projects: A Systematic Review. *IFIP Adv. Inf. Commun. Technol.* 47 (2014), 153-163, Springer Berlin Heidelberg.
22. Watson, C., Li, F.W.B., and Godwin, J.L. BlueFix: Using Crowd-Sourced Feedback to Support Programming Students in Error Diagnosis and Repair. In E. Popescu, Q. Li, R. Klamma, H. Leung, and M. Specht, eds., *Advances in Web-Based Learning - ICWL 2012*. Springer Berlin Heidelberg, 2012, 228–239.