Crowd Development

Thomas D. LaToza[1], W. Ben Towne[2], André van der Hoek[1], James D. Herbsleb[2]

[1]Department of Informatics
University of California, Irvine
Irvine, CA USA
{tlatoza, andre}@ics.uci.edu

[2]Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA USA
{wbt, jdh}@cs.cmu.edu

***Abstract*—Crowd development is a development process designed for transient workers of varying skill. Work is organized into microtasks, which are short, self-descriptive, and modular. Microtasks recursively spawn microtasks and are matched to workers, who accrue points reflecting value created. Crowd development might help to reduce time to market and software development costs, increase programmer productivity, and make programming more fun.**

***Index Terms*—crowdsourcing, distributed development, gamification, social software development**

I. Introduction

Today's crowdsourcing systems enable large, challenging tasks to be performed in parallel by crowds of transient workers. In 2011, players of the game Foldit produced an accurate 3D model of an enzyme in just 10 days, a problem that had stumped researchers for 15 years [7]. A team in the DARPA network challenge recruited a crowd to locate 10 balloons, placed throughout the United States, in under 9 hours. And while still in beta, over 100,000 players of Duolingo translated websites into a foreign language as they learned a language with their friends [3]; that project continues to grow.

One of the earliest successes of the crowd is open source software development: large, reliable, open source projects such as Linux and Apache demonstrate the scope of what crowds can achieve. However, open source software development still presents a fairly high barrier to participation for activities beyond bug reporting, with mailing lists to join, a codebase to explore, conventions to learn, a distributed social network of various personalities to navigate. This dissuades developers who are not strongly committed from contributing, leaving a large amount of the potential value in the "long tail" of participation untapped.

We believe the time has come to fundamentally re-envision software development for the crowd. What if a transient worker could join a project and contribute immediately? What if workers learned valuable software development skills while contributing? What if software development felt more like a game, and less like work? What if software development work supported fine-grain parallelism, so that a crowd could build a large application in a day?

Crowd development is a development process that organizes work into *microtasks*. Microtasks are short – on the order of seconds to a few minutes – and provide specific completion criteria. Microtasks are self-descriptive, enabling workers to immediately contribute. Microtasks are modular, enabling fine-grained parallelism by providing a view of the system one artifact (e.g., method) at a time, exposing only aspects of other artifacts needed for the task at hand. Microtasks may fail, through malicious work, disappearing workers, or legitimate error. Microtask completions accrue points in an incentive system, incentivizing good behavior. And microtasks recursively spawn microtasks, allowing the system to detect and advertise work items. Crowd development differs from both traditional and open source development in making the unit of work short, modular, and self descriptive, potentially enabling new techniques for task assignment, feedback, incentives, coordination, and management that better match the work to be done.

This paper describes a vision for crowd development, outlining benefits it might achieve and research challenges to be addressed. How is work divided up? How are tasks matched to workers and quality ensured? What motivates workers to participate? For each challenge, we discuss important tradeoffs and considerations and survey potential solutions, recognizing that there may be many valid alternatives.

II. Related work

As 'outsourcing' is the practice of taking jobs traditionally done by experts in one company in one region and having them be done by different individuals in a different region, 'crowdsourcing' is the practice of enabling a large, distributed, crowd to complete work traditionally done by experts within one firm. Several reviews provide an introduction to this field, suggesting that a large diverse crowd will generally do a better job than a single expert, and describe some conditions under which that is more likely to be the case [6][16][19]. Much work in the area of crowdsourcing has focused on techniques for ensuring quality, such as using redundancy or filtering out work done by unskilled or unmotivated workers. A number of experiments have helped build theory and practices that allow tasks with individual accuracy below traditionally acceptable levels (e. g. 70% correct) to be combined in ways that lead to overall task quality above even high quality thresholds. Further work in crowdsourcing has examined construction of complex workflows [10] [16], so that even creative work or tasks with nontrivial interdependencies can be crowdsourced. This work provides essential building blocks for crowd development.

A few systems have begun to explore crowdsourcing individual software development tasks. TopCoder organizes competitions for tasks such as authoring algorithms, UI, and software design [20]. Collabode provides an experience similar to Google Docs for code, enabling developers to "micro out-

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

source" freeform tasks to other developers [5], and MobileWorks plans to add programming tasks [9]. But no system has yet scaled crowdsourcing from development tasks to a development process.

III. Re-envisioning Development for the Crowd

Crowd development might enable a range of benefits that fundamentally improve the way software is built.

Shorter time to market: By dividing tasks into microtasks, crowdsourcing increases the potential for parallelism. Work that might have been done by a single worker over an hour might now be done, in parallel, by 20 workers in three minutes. Taken to the extreme, this could enable dramatically faster time to market. If a crowd of 1,000,000 workers worked on a single application for a day, could it build a large application?

Democratizing development: Players of the game Foldit fold proteins not through their expert knowledge of biology, but through a game that encodes it into the rules. Recent work has applied this paradigm to verification, encoding the construction of formal specifications as a puzzle game for non-programmers [15]. Fragmenting software development work into microtasks opens the possibility of allowing non-programmers to contribute either directly or through re-encoding the work required by the microtask as a game. Could a game that encodes writing sort routines into a puzzle be a grandmother's past time?

Increased programmer productivity: Studies have long found that some developers are more than 10x more productive than others [17]. One source of these differences is knowledge: expert developers see code in ways that help them work far more effectively [12]. For example, in debugging, having seen a similar bug before can have a greater effect than a dedicated tool [11]. The ability to search the web for code snippets and explanations of errors has begun to more efficiently distribute knowledge from the haves to the have-nots. But programmer productivity might be even further increased by designing mechanisms to share a wider range of knowledge and more effectively matching questions to experts.

Reduced software development costs: Crowdsourcing systems such as Duolingo provide a new paradigm for compensating workers, providing a fun, intrinsically motivating experience that delivers value to them as a byproduct of their work. Compared to micropayment compensation models such as Mechanical Turk and MobileWorks, this is in some sense a "fairer" model, where individuals gain the value of learning another language, while translating the Web, for free. This also means much lower cost for the project sponsor. Translating the remainder of the English Wikipedia into Spanish using professional services, even at low rates of ($0.05 / word), would cost over $50M (p. 53 [13]), but may be done for nearly free in Duolingo. This two-sided generation of value could be applied to software, dramatically reducing the costs of developing new software and opening new capabilities for nonprofits that cannot afford the costs of either traditional contract software developing or the costs associated with cultivating a new open source community.

Making programming more fun: Programming is a challenging, skilled endeavor that many may sometimes enjoy. But programming also involves frustrating debugging and writing tedious boilerplate code. Studies have found that a task is fun when, moment to moment, it is optimally challenging, balancing the demands of the task to the skill of the worker so that it is neither too challenging to prevent progress nor too easy to be tedious [1]. Organizing work into small, self-descriptive, microtasks so that anyone can quickly begin work, makes it easier for a bored or frustrated developer to skip to the next task, letting the system find another developer that might be better suited to the work. Moreover, more finely partitioning tasks permits workers to specialize and request tasks that are better match to their own unique interests and skills.

Conducting software engineering field experiments: Companies such as Google rigorously test potential new user experiences with A/B testing, a user evaluation technique in which portions of the user population are automatically assigned to a control or experimental condition, and performance measures are taken. If users notice at all, it is only that they were seamlessly migrated to a new version; no recruiting of participants is necessary. Crowd development enables software engineering researchers to adopt this approach by shifting the programming environment from the desktop to microtasks in the cloud. Software engineering researchers could then evaluate new tools and ideas (e. g. about task dependency structures) through experiments conducted with real users doing real work. Moreover, working in the cloud on small, individually evaluated microtasks provides a range of easily captured, fine-grained measures of task time and success. A/B testing could dramatically lower the barriers to experimentation in software engineering, greatly increasing the available evidence for the usefulness of new tools and practices.

IV. Division of Labor

A. How Can Development Work be Effectively Decomposed?

Can software development work be split into microtasks that are only seconds or a few minutes in duration? Can a software development microtask be done by a transient worker with no knowledge of the project? Can a worker reason completely modularly about a method? What knowledge is required to do development microtasks? Can some tasks be expressed as puzzle games playable by non-programmers?

What might a microtask look like:

Sketch a method. Workers are given a description of a funciton, its signature, and pre-and post conditions, and asked to sketch pseudocode of an implementation. When workers have questions (e.g., "how can I make this string uppercase?", or "is this parameter guaranteed to always be a positive number?"), they ask the crowd. When a worker wants to call a function, they describe what it should do. Microtasks are then generated to expand the description, find and reuse an existing function if present, or write a new implementation iteratively.

The Testing Game. Players write input and output pairs for a function. The actual output of the function is then revealed, and players are informed if their output matches. If it does not, players have a choice: bet some of their points that they are correct or update their answer to match the function. If they choose to bet, additional workers are independently shown the

two outputs and asked to vote on the correct value, without knowledge of each output's source. If the crowd votes for the worker's value, the worker wins their bet, and the failing test generates a debug microtask.

Debug a function. A worker sees a method and a failing unit test, and is asked, can you fix the code to make it pass? When values returned by functions appear wrong, the worker describes the erroneous behavior, spawning microtasks to write a new unit test and fix the function.

B. How Can a Crowd be Coordinated?

How can a crowd create a system with conceptual integrity? What interdependencies between microtasks must be managed? How can a crowd make decisions? How can knowledge be presented to transient workers who do not know the right question to ask? A range of coordination models might help to address these challenges:

Collective decision making: A worker poses the question, "How should I store this data?" One set of workers – knowledge librarians – scour the answers to existing questions, looking for an answer. If none is found, a second set of workers – database experts – each independently answer the question. Finally, a third set of workers – decision makers – rate each answer and the system picks the highest rated option.

Pushing knowledge: A worker reviews implementations containing an interaction with a database, checking to ensure that it is compliant with each decision in the library of database interaction decisions.

Iterative critique: A crowd of workers is each given a description of a webpage element to design. Each worker edits html and css (by code or WYSIWYG) to create an initial design. A second crowd then visits a gallery of designs, adding critiques expressing both positive and negative aspects and assigning an overall rating. Ratings are tabulated, the best design chosen, and critiques returned to the designers. A new crowd is then tasked to improve the artifact, addressing the criticisms and incorporating positive aspects from the other designs. Iterative critiques continue until the crowd is collectively satisfied with the final product.

C. What Makes a Decomposition Efficient?

The decomposition of software development work can be characterized along a number of dimensions, such as the granularity of the system view, the nature of sequential dependencies and flow of information between microtasks, the number of alternatives solicited, and the mechanisms for aggregating conflicting alternatives. Efficient decompositions increase parallelism (reducing time to market), minimize overhead and rework, effectively distribute knowledge, and ensure quality. Many of these objectives may conflict, leading to different crowd development approaches optimized for different qualities.

V. Assigning Work & Ensuring Quality

A. How Should Microtasks be Matched to Workers?

Effectively matching workers to microtasks requires matching the skill and knowledge demands of the task to the abilities and interests of the worker, while simultaneously ensuring work does not stall waiting for microtasks to complete. There are three general approaches to task assignment: workers select microtasks, the system assigns microtasks, or workers assign microtasks to other workers. In traditional organizations, task assignment is often done by other workers (e.g., managers or bug triagers). In open source projects and existing crowdsourcing systems such as Mechanical Turk, task assignment is often done by the workers themselves, who decide what interests them, as letting workers choose their work enhances intrinsic motivation [8]. Any of these designs could be used in crowd development. However, manual task selection, by the workers themselves or other workers, adds overhead, which is particularly acute due to the small size of the microtasks. A worker who spends 1 minute looking for a 1-minute microtask is half as productive, all else being equal.

It may be possible to provide the motivational benefits of choice while enjoying the efficiency benefits of automatic task assignment. Workers could express task preferences (e.g., write tests, debug), which the system uses to generate assignments. This provides a rich space for exploring the attributes of the task and worker critical to generating the best match.

B. How Can Quality be Ensured?

How can quality software be produced by workers that come and go, are sometimes malicious, and are often mistaken? A key approach is to design redundancy into the task decomposition, so that multiple alternatives are generated and compared, artifacts are checked against other artifacts, or artifacts are reviewed. But reviews may themselves be erroneous. In this case, a worker might challenge a review, generating new independent reviews, which is ultimately aggregated into a new review. Heavy emphasis on unit testing may also increase code quality.

C. What Incentivizes Good Behavior?

How can workers be motivated to work on an unpopular microtask or invest extra effort to achieve long-term benefits? To encourage workers to do challenging microtasks, points in an incentive system should reflect its difficulty, as measured by responses by other workers and similarity to other microtasks. To encourage quality work, points should reflect the value created both directly and indirectly, directly measuring quality and quantity and attributing later value created (e.g., a function reused) back to the responsible developers.

VI. Motivating the Crowd

What might motivate a worker to join and contribute to a crowd development project? Studies of open source projects suggest many potential motives: the desire to learn and develop new skills, to share knowledge and skills, to improve F/OSS products, and to participate in a new form of cooperation [4]. As in F/OSS projects, volunteers, paid employees of a firm, or even contract programmers might constitute crowd development projects. For employees, the incentive system might provide a wealth of fine-grained information about an employee's skills and contributions to be used in performance reviews.

As in games such as Foldit and Duolingo or more generally in the gamification of work [2], crowd development might also be organized as a game. Most games involve a task and

achievements. Microtasks provide a task; the assignment algorithm attempts to create an optimal level of challenge. Gaining points and reaching particular achievements might provide internal and/or external reputational benefits, as well as access to the most challenging and important microtasks. Games also rely on conflict, competition, strategy, and the possibility of failure to create interest and fun [14]. Workers might compete to produce the best alternative, bet on the correctness of their answer, or race the clock. Of course, workers may strategize to exploit the system, and care is needed to ensure that the worker's and system's goals are aligned.

Microtasks might also provide educational value, transferring knowledge from experienced to the less experienced. Some developers find it helpful to watch expert developers work, as reflected in websites that webcast programming sessions. Workers might similarly be allowed to observe, compete, or collaborate by pairing with another developer.

VII. Preliminary Work

We are currently exploring the feasibility of crowd development through the construction of a prototype system. Our initial focus is in answering the central question of how software development might be decomposed into microtasks. In our prototype, a worker logs in to a web app, which fetches an available microtask. Each microtask consists of a single page containing any needed instructions, descriptions, editors, and other artifacts required to complete the work. Decomposition occurs through an iterative top-down programming methodology, decomposing uses cases into entrypoints, function descriptions, function implementations, and tests, which may each iteratively recurse. Microtask completions transition the state of each artifact, which may then cause new microtasks to be generated. By testing our prototype in practice, we will iteratively improve its design from our experiences.

VIII. Conclusion

Crowd development envisions a software development process optimized for sharing knowledge, distributing work efficiently, motivating contributions, and ensuring quality. But envisioning development structured as self-contained microtasks done by a transient crowd introduces a host of research challenges. Crowd development may not be well-suited to all domains, such as those that require large amounts of domain expertise, safety critical systems, or those with sensitive business information. And how crowd development compares to other development processes ultimately depends on the success of meeting these challenges. But many also reflect important, fundamental questions about software engineering, whose answers might lead to better empirical theories of the nature of software engineering work and a better understanding of such core concepts as modularity, expertise, task interdependencies, and coordination. Many of these challenges are also related to topics in social and organizational psychology, the study of games, and other fields; this provides an opportunity to use results from these fields to re-envision software engineering. While efforts in crowdsourcing development tasks and social software development have begun to explore some of these issues, we believe there remains a large and important space to explore to scale crowdsourced development tasks to crowdsourced software development.

References

[1] M. Csikszentmihalyi. Flow: The Psychology of Optimal Experience. Harper and Row, 1990.

[2] S. Deterding, D. Dixon, R. Khaled, and L. Nacke. "From game design elements to gamefulness: defining gamification." In 15th International Academic MindTrek Conference: Envisioning Future Media Environments (MindTrek), 2011, pp. 9-15.

[3] "Duolingo," duolingo.com, 11/2/12.

[4] R. A. Ghosh, "Understanding free software developers: findings from the FLOSS study." Making Sense of the Bazaar: Perspectives on Open Source and Free Software, J. Feller, B. Fitzgerald, S. Hissam and K. Lakhani (eds.), MIT Press, 2005.

[5] M. Goldman, G. Little, and R. C. Miller, "Collabode: collaborative coding in the browser." Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), 2011. pp. 65-68.

[6] J. Howe, Crowdsourcing: Why the Power of the Crowd Is Driving the Future of Business. Crown Business, 2008.

[7] F. Khatib, S. Cooper, M. D. Tyka, K. Xu, I. Makedon, Z. Popović, D. Baker, and Foldit Players, "Algorithm discovery by protein folding game players," Proceedings of the National Academy of Sciences, vol. 108, 2011.

[8] R. E. Kraut, P. Resnick, with S. Kiesler, Y. Ren, Y. Chen, M. Burke, N. Kittur, J. Riedl and J. Konstan. Building Successful Online Communities: Evidence-Based Social Design. MIT Press, 2012.

[9] A. Kulkarni. "Next-generation crowdsourcing platforms." Pittsburgh, PA: Crowdsourcing Lunch, 24 October 2011.

[10] A. Kulkarni, M. Can, and B. Hartmann, "Collaboratively crowdsourcing workflows with turkomatic." Computer Supported Cooperative Work (CSCW), 2012, pp. 1003-1012.

[11] T. D. LaToza, "Answering reachability questions." Dissertation, Carnegie Mellon University, 2012.

[12] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers, "Program comprehension as fact finding." In ESEC-FSE, 2007, pp. 361-370.

[13] E. Law and L. von Ahn. Human Computation. Morgan & Claypool Publishers, 2011.

[14] J. Lekevicius. "Game design – what makes games fun and addictive." Sept 2010, www.slideshare.net/flixic/game-design-what-makes-games-fun-and-addictive.

[15] W. Li, S. A. Seshia, and S. Jha, "CrowdMine: towards crowdsourced human-assisted verification." Design Automation Conference (DAC), 2012, pp. 1254-1255.

[16] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller, "TurKit: human computation algorithms on mechanical turk." In UIST, 2010, pp. 57-66.

[17] S. McConnel, "What does 10x mean? Measuring variations in programmer productivity." In Making Software, O'Reilly, 2011.

[16] S. E. Page, The Difference: How the Power of Diversity Creates Better Groups, Firms, Schools, and Societies. Princeton University Press, 2008.

[19] J. Surowiecki, The Wisdom of Crowds. Random House, Inc., 2005.

[20] TopCoder. www.topcoder.com.