A Vision of Crowd Development

Thomas D. LaToza and André van der Hoek
Department of Informatics
University of California, Irvine; Irvine, CA, USA
{*tlatoza, andre*}*@ics.uci.edu*

***Abstract*—Crowdsourcing has had extraordinary success in solving a diverse set of problems, ranging from digitization of libraries and translation of the Internet, to scientific challenges such as classifying elements in the galaxy or determining the 3D shape of an enzyme. By leveraging the power of the masses, it is feasible to complete tasks in mere days and sometimes even hours, and to take on tasks that were previously impossible because of their sheer scale. Underlying the success of crowdsourcing is a common theme — the microtask. By breaking down the overall task at hand into microtasks providing short, self-contained pieces of work, work can be performed independently, quickly, and in parallel — enabling numerous and often untrained participants to chip in. This paper puts forth a research agenda, examining the question of whether the same kinds of successes that microtask crowdsourcing is having in revolutionizing other domains can be brought to software development. That is, we ask whether it is possible to push well beyond the open source paradigm, which still relies on traditional, coarse-grained tasks, to a model in which programming proceeds through microtasks performed by vast numbers of crowd developers.**

***Index Terms*—Crowdsourcing, collaborative software development, open source software development**

I. Introduction

Crowdsourcing has demonstrated a wide range of successes in enabling large, challenging tasks to be performed quickly by massive crowds of untrained, casual 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 [10]. Over 10 million people use Duolingo to learn a language by translating small snippets of text; aggregating these translations produces translations of websites and other documents [2]. Building on the broad use of Amazon Mechanical Turk, firms such as MobileWorks[1] and CrowdFlower[2] provide a managed environment to enable clients to complete often urgent work with a crowd [14].

Underlying this success is a common theme — the microtask. Microtasks are short and self-contained, exploiting the "long tail" of casual contributors to enable numerous and often untrained participants to quickly chip in. Microtasks enable parallelism, allowing large and complex tasks to be completed through massive numbers of small contributions — done in parallel — that are aggregated into a conclusive result. Microtasks enable intelligence in the environment, leading to automatic approaches that actively manage task generation and assignment. This makes possible new workflows soliciting and aggregating diverse ideas, assignment of workers to tasks based on fine-grained expertise, and even the gamification of tasks within larger workflows.

What if this model could be applied to software development? There are many important and high impact situations in which there is a clear and compelling need for software to be built rapidly: when responding to a disaster, or fixing suddenly-apparent deficiencies in a key software system, perhaps in cases of an escalated cyber conflict. A common response is "all hands on deck", mobilizing developers across an organization or community to contribute until the issue is resolved. But traditional development processes are not designed for this mobilization and certainly not at scale, making it challenging to support developers in making small contributions, identify useful tasks, and coordinate the organized chaos of massive ad-hoc work.

While open source development is crowdsourcing, it is not microtasking, as development tasks remain large, workers must be generalists, and participation is large-scale but not massive. Traditional open source indeed imposes barriers to contribution, requiring developers to learn the codebase, identify and install tool infrastructure, socialize into community conventions for contributions, and identify work that might be accepted [7][13]. These factors help dissuade casually committed developers from contributing, leaving a potential "long tail" of contributors untapped.

This paper puts forth a research agenda to explore the decomposition of software development into microtasks, enabling crowds of developers to immediately and effectively contribute by generating, distributing, and coordinating software microtasks. Underlying this agenda are three necessary considerations — decomposition, coordination, and quality — each of which influences and impacts how tools might enable building software with a crowd. In this paper, we consider each of these aspects, examining several challenges raised and potential approaches that might be taken.

II. Examples

How might a developer participate in crowd development? To provide a sense of the styles of work we envision and the challenges these incur, we provide several vignettes of crowd development in practice.

Bob is sitting on a train and feels like programming for a few minutes. Logging in to a website, he sees a couple of interesting projects. Being an avid cyclist, he immediately chooses the one to build a web app for urban cycling maps. He then gets a microtask — to sketch some pseudocode for a function. After finishing, he decides he is in the mood for some testing and sets his task selection preferences to "write test cases". Seeing a description and signature of a function and a list of typical is-

[1] www.mobileworks.com
[2] www.crowdflower.com

sues to test, he writes a list of test cases, labeling each with an issue to describe its purpose. He realizes one of these represents a new issue that might be of interest to others, so he adds it to the issue list. Each time he logs in over the next week, he is excited to see others using his issue, adding to his points.

Julie is an experienced software architect who just received an email. Recalling that she signed up to donate time to a crisis response group, she sees that a disaster has just occurred and that the group needs to respond rapidly and build a website linking the local technical infrastructure of the first responders to the disaster response needs as fast as possible. Logging in, she sees some architectural questions other developers have asked. After asking her own questions to track down related decisions and getting some responses, she answers each question. She then receives a microtask to moderate suggested answers to another architectural decision. Reading each option, she up or down votes each, adding a comment describing some important caveats to one.

Ting is a freshmen biology student. After completing a few Codecademy classes, he is eager to write code within a real software project. Logging in, he picks an interesting looking healthcare project. He first receives a debugging task. Seeing a failing test and a function, he tries a few things he hopes might make the test pass. But nothing works, so he skips to another task. This time, Ting receives some pseudocode and works to implement the function. But the pseudocode deals with a complicated graph traversal algorithm, and he has never seen anything quite like it, so he skips that too. The system then asks if he would like to watch an expert work. Being a little lost, he agrees. As he watches the expert work on several microtasks, he follows the chat window on the side as several other workers try to explain what the expert is doing. Even the expert chimes in here and there with some explanations. After feeling like he is ready to try writing some code again, he begins implementing some pseudocode again. A little while later, he gets a review back, explaining some things he did well and providing a few suggestions for improvement.

III. Research Agenda

How can such a vision of crowd development be realized? In our previous work, we surveyed several ideas for crowdsourcing software development [17] and developed a prototype online IDE for microtasking simple programming tasks [16]. Yet, many challenges remain: decomposing a broad range of software development work into microtasks, coordinating contributions at the scale of crowds, and ensuring the quality of the software produced by the crowd. In this section, we examine these challenges and present an agenda articulating how these challenges may be met.

A. Decomposition

How can developers contribute to a software project in small, self-contained microtasks? What will these microtasks ask developers to do, and how can this work be aggregated to complete a larger task? What context is required to perform the task, and how can this information be provided? What portions of software development work can be decomposed and made parallel, and which portions are inherently sequential? How do different approaches to decomposition or choices of microtask boundary affect the ways in which microtasks can be made short, self-contained, and parallelizable, and how is this influenced by the type of work to be done? A number of decomposition approaches might be possible, incorporating iterative or hierarchic workflows in different ways. A key aspect is one of granularity: smaller microtasks enable greater parallelism, reducing clock time, but may increase communication overhead.

One way in which tasks might be decomposed is through artifacts, creating microtasks which each ask workers to perform a single task on a single artifact. Artifacts encompass natural boundaries in software work; of course, there are important considerations in choosing between granularities such as a functions and tests and more coarse-grained decompositions such as classes or modules. Microtasking some tasks may require the invention of new artifact boundaries. For example, when determining how to reuse an external library, developers must read documentation, find examples, customize and experiment, and arrive at a solution. As a fleeting understanding, much of this work may not be captured in the function itself and this knowledge of the library may in any case be valuable for interactions with the library across many artifacts. Reifying this interaction into a new, synthetic artifact — an example and explanation of the use of an API for a specific task — allows discovery of information about the API to be scheduled as separate work, dependencies on artifacts using the library established, and a dedicated micro task interface for library use to be created.

B. Coordination

In opening software development to contributions by the crowd, new challenges emerge in coordinating at scale. How can workers be matched to microtasks, most efficiently allocating the knowledge workers bring to bear to the work to be done? Which aspects of software work benefit most from expertise, and how can this expertise best be leveraged? How can a system track the work to be done, and automatically generate microtasks to perform next? How can the work of many microtasks, each concurrently potentially changing the artifacts, be coordinated? How are dependencies between work detected and managed? What aspects of software engineering work can be done with only local information, and what aspects require a larger global view?

Inherent in software work are concerns that crosscut the artifact structure, leading to their scattered implementation in diverse artifacts across a codebase [11]. Much of system design has this character, as higher-level decisions as to how requirements are achieved ultimately influence lower level decisions across a codebase. Our studies of program comprehension suggest that developers working at a code level perceive design as a network of decisions, as they explicitly reverse engineer decisions in code and attempt to understand dependencies on these decisions that act as constraints [15]. In higher-level design, theorists of socio-technical systems have conceptualized software architecture as a network of decisions [3][5], embodied in notations such as a design structure matrix [19]. This suggests a tantalizing question: can a singular design with conceptual integrity be created — in parallel — as individual decisions that are coordinated through their dependencies?

This idea brings several challenges, drawing on fundamental questions about the nature of design in software. First, how can dependencies be identified; how can a worker editing code

discover relevant decisions? One approach might be to enable workers to ask a question to find information they need (e.g., what is the right way to serialize data into the data store?), which is then matched against existing decisions, generating a new decision to be made if none is found. Yet developers are unlikely to always realize what they are doing is relevant to a decision. Another approach might be to review code for its conformance to decisions, tasking a worker to review a function for conformance with a short checklist of related decisions. Enabling workers to specialize in these reviews (as with any microtask) might allow them to be performed quickly, perhaps even by allowing workers to routinize their inspection steps through the use of automated scripts (e.g., calls to serialization methods), generating suspicious methods to be inspected. Specialization and contributor-written scripts to automate certain microtasks is common in existing crowdsourcing communities such as Wikipedia and Foldit.

Also necessary are mechanisms for generating ideas, identifying dependencies on existing decisions, understanding constraints, debating tradeoffs, and ultimately producing a decision. Interestingly, Q&A sites such as StackOverflow[3] have much of this character, as a question is posed, answers are crowdsourced, votes are cast, and the requestor picks a winning answer. Could this model be used to make important design decisions? Quirky[4] and Assembly[5] seem to demonstrate that it can, at least in the context of product design.

C. Quality

A fundamental challenge in crowdsourcing systems lies in the use of contributions from the masses to produce quality work. Workers may do too little, act maliciously, or even be "eager beavers" who do more than the system intended [4]. In response, crowdsourcing systems have explored the use of explicit reviews by requesters (e.g., Mechanical Turk) and techniques for aggregating redundant work (e.g., games with a purpose [1], the map/reduce paradigm [12]). Underlying these approaches is a fundamental insight: through the "wisdom of the crowd", a large group's independent, redundant solutions can be as good as, and often better than, those of any individual member [20]. How can redundancy be effectively utilized in software development work to promote quality? What aspects of software engineering work are most important, and might benefit most from the high quality — but expensive — work done by large-scale redundancy? How can small-scale redundancy — tasks done by a few — be automatically aggregated to produce higher quality work? In what situations is redundancy a more effective way to achieve quality than through reviews? An important issue arises in how work is assessed: as work is evaluated at small scale, how can the longer-term implications of work also be evaluated and incentivized?

Many traditional solutions — such as voting on independent redundant solutions — are unlikely to be applicable, due to the greater diversity of valid responses possible in software work. Thus, new approaches to achieving quality must be found or adapted to microtasking software work. One approach to quality is to use outcome-based incentives, measuring positive events (e.g., a test catching a bug) and apportioning some of the value created back to its producers through individualized incentivizes. As events occur in the system — tests leading to bug fixes, functions being reused, a worker beginning to produce higher quality work — credit is apportioned back to workers responsible for creating this value — a worker that wrote a test, a well defined function interface, constructive feedback provided in a review. Incentives may then be translated into appropriate rewards, depending on the context, such as small payments, public displays of reputation such as badges and points, and access to more interesting and prestigious work. The key challenge underlying this approach is in finding appropriate measures to value work.

Another approach to ensuring quality is through workflows that incorporate redundancy or reviews. There are a range of applicable approaches, each with important tradeoffs. Soliciting a single contribution and review for each microtask is simplest, but may not result in a particularly high quality output. Independently soliciting many redundant contributions and selecting the best provides a potentially higher quality solution at a greater invested effort. Sequentially soliciting iterative contributions to an artifact — e.g., generating new microtasks to edit a function until all the pseudocode has been implemented — may make effective use of worker's varied expertise and ability to contribute, but requires effective ways to value each edit to prevent social loafing. Each of these approaches can be combined and nested to create more complex workflows, potentially exposing tradeoffs between cost and quality which can be used to ground informed decisions about system design. For example, implementing a function might involve first starting with five independent solutions, each of which are iteratively evolved, and then compared to select the best. In these situations, it is important to gauge the relative importance of a work product before it is created, to understand how much effort it may be appropriate to invest in its creation. Microtasks that have a strong influence on the subsequent work to be done are important to invest in. For example, a microtask to implement a function at the root of a large algorithm particularly effectively might reduce the amount of other functions that must be created. But how can importance be predicted? Workers requesting work (e.g., writing a pseudocall requesting a function) might be asked to rate its importance; or microtasks that workers have chosen to skip might be inferred to be challenging.

IV. Key Challenges

Many issues inherent to crowd development crosscut consideration of decomposition, coordination, and quality. Chief among them are two central considerations: the tradeoffs between microtasks and context and between modularity and coordination.

Microtasks enable transient workers to contribute in small ways by being short, enabling vast parallelism and speed through small contributions. Yet, workers must still have enough context and background to get the work done. This requires careful attention to microtask design, balancing the need for making microtasks small against the need for them to be self-contained. Where this balance lies for different software

[3] www.stackoverflow.com
[4] www.quirky.com
[5] assembly.com

development tasks is a fundamental question, and likely will only be understood through experimentation and trial and error.

A similar issue is at play in coordinating software work. More visibility of ongoing work may make it easier to coordinate and manage dependencies amongst microtasks. But the more developers must understand and relate their microtask to other ongoing work, the less microtasks remain self-contained. The key question is, again, how to balance this tradeoff in designing microtasks.

Thus, a core consideration in microtasking software development work is one of information needs: exactly what information do developers need to do software development tasks in-the-moment, and how can work be decomposed and organized into microtasks, supported by the environment, that reflect the natural structure of information needs in software development work?

V. Related Work

Beyond open source software development, there are a number of ways in which crowdsourcing has begun to be applied to software development. TopCoder[6] enables programmers to participate in competitions over the course of hours or days, competing to implement features, build a UML diagram, or find bugs. HelpMeOut [9] lets novice programmers share their fixes to common programming bugs. Learn-to-program sites such as Codecademy scaffold learning through a manually curuated series of microtasks. Even StackOverflow's basic model is one microtasking: developers ask questions, other developers answer them, and yet other developers evaluate the quality of the answers. Micro-outsourcing enables a developer to request small tasks to be done by the crowd [8]. Non-programmers are increasingly being brought to work on software projects. While beta testers have long provided feedback, sites such as uTest[7] and trymyUI[8] make the process more systematic and explicit, enabling users to be recruited to rapidly provide feedback on specific issues. Other work has sought to transform software development tasks into games for non-programmers [18][6], such as Pipe Jam [6], which transforms authoring formal specifications into a puzzle game.

VI. Conclusions

Crowd development envisions a new way in which to build software, encompassing transient, fluid workforces automatically arranged by the environment to perform microtasks within a workflow. As in any potentially disruptive idea, it is far from clear in what contexts, if any, it may ultimately prove its value. But in exploring questions such as what context and information is required by developers in microtasks, the exploration itself may create important new scientific knowledge about the nature of software development work, which may be broadly valuable in many ways.

Acknowledgments

This paper is a summary of the NSF grant CCF-1414197, by which it is also partially supported.

[6] www.topcoder.com
[7] www.utest.com
[8] www.trymyUI.com

References

[1] L. von Ahn and L. Dabbish, "Designing games with a purpose," CACM, 51(8), 2008, pp. 58-67.

[2] L. von Ahn, "Duolingo: learn a language for free while helping to translate the web," IUI 2013, pp. 1-2.

[3] C. Y. Baldwin and K. B. Clark, Design Rules: The Power of Modularity. MIT Press, Cambridge, MA, 1999.

[4] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich, "Soylent: a word processor with a crowd inside," UIST 2010. pp. 313-322.

[5] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," TSE, 35(6), 2009, pp. 864-878.

[6] W. Dietl, S. Dietzel, M. D. Ernst, Nathaniel Mote, Brian Walker, Seth Cooper, Timothy Pavlik, and Zoran Popović, "Verification games: making verification fun," Workshop on Formal Techniques for Java-like Programs, 2012, pp, 42-49.

[7] N. Ducheneaut, "Socialization in an open source software community: a socio-technical analysis," CSCW, 14(4), 2005, pp. 323-368.

[8] M. Goldman, G. Little, and R. C. Miller, "Collabode: collaborative coding in the browser," CHASE 20111, pp. 65-68.

[9] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: suggesting solutions to error messages," CHI 2010, pp. 1019-1028.

[10] 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," PNAS 2011. . **????**

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, "Aspect-oriented programming," ECOOP 1997, pp. 220-242.

[12] A. Kittur, B. Smus, S. Khamkar, and R. E. Kraut, "CrowdForge: crowdsourcing complex work," UIST 2011, pp. 43-52.

[13] G. von Krogh, S. Spaeth, and K. R. Lakhani, "Community, joining, and specialization in open source software innovation: a case study," Research Policy, 32(7), 2003, pp. 1217-1241.

[14] A. Kulkarni, P. Gutheim, P. Narula, D. Rolnitzky, T. Parikh, and B. Hartmann, "MobileWorks: designing for quality in a managed crowdsourcing architecture," Internet Computing, 16(5), 2012, pp. 28-35.

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

[16] T. D. LaToza, W. B. Towne, C. M. Adriano, and A. van der Hoek, "Microtask programming: building software with a crowd," UIST 2014, pp. 43-54.

[17] T. D. LaToza, W. B. Towne, A. van der Hoek, and J. D. Herbsleb, "Crowd development," CHASE 2013, pp. 85-88.

[18] W. Li, S. A. Seshia, and S. Jha, "Towards crowdsourced human-assisted verification," DAC 2012, pp. 1254-1255

[19] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," ESEC/FSE 2001, pp. 99-108.

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