Harnessing the Crowd: Decontextualizing Software Work

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

ABSTRACT
Organizing software work into self-contained, low-context microtasks opens new opportunities for software development, reducing the barriers to contribute to software work and enabling software projects to be more fluid. Achieving this vision requires understanding the role of context in software development and designing new approaches for managing context.

Categories and Subject Descriptors
D.2.3 [**Software engineering**]: Coding tools and techniques

Keywords
crowdsourcing, context in software development, open source software development

1. INTRODUCTION
Software development has traditionally required developers to first learn the context of their work before they can effectively contribute. Developers must know context such as where features are implemented, how to implement changes consistent with an architecture and design, and which developers to ask questions. As a result, developers engage in a process of onboarding, learning the codebase and building a mental model of its architecture, design, and implementation.

Recently, a number of trends under the broad banner of crowdsourcing have begun to demonstrate that not all software development work requires such context. For example, developers on Q&A sites such as StackOverflow[1] answer questions with only the context in the question itself, helping developers who might once have created a code snippet by programming to instead simply ask a question. In effect, a requesting developer may crowdsource subtasks to developers on a Q&A site, who then perform software development work with only the context explicitly provided by the requestor. Beyond Q&A sites, this crowdsourcing paradigm has also been explored in competition sites such as TopCoder[2] and testing sites such as uTest[3]. We refer to crowdsourcing approaches that are short and that require little or no context as *microtasking*, differentiating them from approaches for open contribution requiring context such as open source software development. A microtask is a short, self-contained task providing a worker a specific completion criteria (e.g., answer a programming question).

Decontextualizing software work enables microtasking to transform the nature of work, greatly expanding the pool of potential workers from the small number of developers who are members of a project to the millions of developers participating in crowdsourcing platforms. Experts, specialists in the immediate problem at hand, can then be more easily brought to bear, enabling, for example, a developer who already debugged a similar exception to share their solution on a Q&A site. Decontextualizing work also reduces contribution barriers to projects, enabling transient workers who might otherwise be unable to contribute to help out and potentially enabling work to be accomplished more quickly through teams that are, when necessary, far larger. For example, questions about expert topics posed to StackOverflow are answered, on average, in just 11 minutes [6].

Harnessing experts and speeding work have clear benefits across software development work. But how broadly is microtasking applicable to building software? The key barrier to achieving this vision is understanding the context required to perform tasks. Enabling casual, transient, work requires microtasks that are self-contained and that decontextualize the work by embedding the necessary context into the task itself.

To explore approaches for embedding context into microtasks, we have designed an online IDE for microtask programming, CrowdCode [5], building on earlier efforts to microtask programming such as micro-outsourcing [2]. In CrowdCode, workers simply login to the platform, are given a self-contained microtask containing relevant information, and can start contributing. In this paper, we explore approaches for decontextualizing programming, debugging, and design.

2. DECONTEXTUALIZING WORK
2.1 Programming
Programming tasks require many types of context. When given a feature to implement, developers must know where to implement it. When writing a function, developers must understand the context in which it is used. When calling a function, developers must understand what assumptions it makes the system's state and the effects that it may cause.

Our hypothesis is that much of the context required in common programming tasks can be captured in the interfaces of functions, enabling tasks to be performed *modularly* on functions in isolation. In some sense, this is the central assertion of design by contract [7]. However, in our work we aim to explicitly test the limits of this idea, providing developers only a single function, in isolation, and *requiring* workers to communicate context only through interfaces between functions.

It quickly became apparent that an additional restriction was necessary: requiring code to be functional. One of the primary challenges developers face in investigation and debugging tasks is to traverse control flow paths through the code [4]. Many of these situations are caused by the necessity to understand *effects*, actions taken in functions that change mutable state or that impact the environment in which a program executes (e.g., redrawing the screen). For example, one developer spent 83 minutes understanding where and how, within a complex set of functions, a data structure was being mutated [4]. While there is yet vigorous debate as to the ultimate benefits of functional programming, requiring programs to be functional seems to reduce the context

[1] www.stackoverflow.com

[2] www.topcoder.com

[3] www.utest.com

necessary to program by eliminating effects and enabling functions to be fully described by their inputs and outputs.

Within this scope, we have explored the possibility of self-contained microtasks for programming. For example, developers may receive an *Edit function* microtask containing a description of a function and be asked to implement it. While writing code, developers may simply request a function by describing its desired behavior through a *psuedocall* (Figure 1), which is then passed to the crowd to either locate a matching function or write a description for a new function. Changing a function description – e.g., adding a parameter – creates microtasks on the function's callers, each informing the worker of the change and asking them to adapt the caller appropriately. In this way, contextual information about what is happening in the rest of the project can be passed along dependencies between functions, potentially enabling a worker to perform tasks in isolation.

2.2 Debugging

When an important bug in a live site is discovered, a common response is, "All hands on deck," mobilizing developers in the project to expeditiously address the problem as quickly as possible. On the one hand, fault localization seems inherently parallel with low context tasks: simply ask workers to, separately, inspect each function for a defect. However, a study found that simply inspecting a code location is often not enough, as developers require richer contextual information [8].

In CrowdCode, we have explored a modular approach for debugging using stubs, using the interface between functions to communicate context (more details of the approach are available elsewhere [5]). When a function fails a test, a *Debug* microtask is generated, providing a worker a code editor with the function's code and a list of failing unit tests. Workers can edit the function and rerun the tests to check if a change has fixed the defect. Of course, the defect may not be in the function itself. In traditional debugging, developers might next be forced to use their contextual knowledge of the codebase to hypothesize locations where the defect might be, making choices about which methods to step in to or investigate. In our approach, developers can instead inspect each function call, viewing the runtime values of each parameter and return value. If a function is not producing a value matching its contract, the return value can be *edited*. This then creates a stub, enabling the worker to continue debugging by rerunning the function's tests, checking if the change has fixed the defect. After the worker submits the microtask, a new test corresponding to the stub is generated and run, which may then generate a new *Debug* microtask on the corresponding function. In this way, workers can debug modularly, relying on the function descriptions and tests to communicate context.

2.3 Design

Design seems inherently a task that that requires a global understanding of a module or software project. How can decisions be made without the availability of context to inform a choice? However, studies of software projects suggest that software designs have structure. One model of design is as a network of decisions, where decisions may have dependencies on other decisions that may affect it [1]. Observations of developers suggest that, when working with complex decisions, developers do not need a global understanding of the entire design. They simply need to understand the rationale underlying the decisions they may be changing [3].

```
different move).

  @param Board board - the initial board prior to the move
  @param Move[] moves - the move(s) to execute
  @return Board - new board
**/
function CRdoMoves(board, moves)
{
    var newBoard = //! copy existing board

    for (var i = 0; i < moves.length; i++)
    {
        if (//! move is a jump
            )
        {
            //! remove piece from the board
        }

        //! create new board with piece moved
        //! check if move created a King

        //# Do we need to do something with checking for victory?
    }
```

Figure 1. Workers may request functions through *pseudocalls* (white background).

We hypothesize that, much as a developer working with a function might use the contracts of other functions to understand its context, a developer working with a decision might use its dependencies to understand its context. Rather than understand an entire project, this greatly reduces the necessary context. Of course, this requires an approach for explicitly managing decisions and identifying their dependencies.

3. CONCLUSIONS

Context is central to software development, being core both to investigation and debugging tasks and to approaches for working more modularly with software. By decontextualizing software tasks into microtasks and requiring developers to work *only* with the information provided, we seek to enable developers to more easily contribute, making it possible to recruit experts for specialized tasks or to more rapidly and fluidly form ad-hoc teams of developers. In exploring this vision, our work may help to reveal situations in which context is crucial and the specific information needs in such situations.

4. ACKNOWLEDGMENTS

This work was supported in part by the NSF under grants NSF IIS-1111446, IIS-1302522, and CCF-1414197.

5. REFERENCES

[1] Baldwin, C. Y., and Clark, K. B. 1999. *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA.

[2] Goldman, M., Little, G., and Miller, R.C. Collabode: Collaborative coding in the browser. In *Proc. of CHASE 2011*, 155–164.

[3] LaToza, T. D., Garlan, D., Herbsleb, J. D., and Myers, B. A. 2007. Program comprehension as fact finding. In *Proc. of ESEC/FSE 2007*, 361-370.

[4] LaToza T. D., and Myers, B. A. 2010. Developers ask reachability questions. In *Proc. ICSE 2010*, 185-194.

[5] LaToza, T. D., Towne, W. B., Adriano, C. M., and van der Hoek, A. 2014. Microtask programming: building software with a crowd. In *Proc. of UIST 2014*.

[6] Mamykina, L., Mannoim, B., Mittal, M., Hripcsak, G., and Hartmann, B. 2011. Design lessons from the fastest q&a site in the west. In *Proc. of CHI 2011,* 2857-2866.

[7] Meyer, B. 1992. Applying "design by contract". *IEEE Computer, 25 (10)*, 40-51.

[8] Parnin, C., and Orso, A. 2011. Are automated debugging techinques actually helping programmers? In *Proc. ISSTA 2011*, 199-209.