Exploring Coordination Models for Ad Hoc Programming Teams

Sang Won Lee
Computer Science & Eng.
University of Michigan
snaglee@umich.edu

Sai R. Gouravajhala
Computer Science & Eng.
University of Michigan
sairohit@umich.edu

Yan Chen
School of Information
University of Michigan
yanchenm@umich.edu

Angela Chen
Computer Science & Eng.
University of Michigan
chenaj@umich.edu

Noah Klugman
Computer Science & Eng.
University of Michigan
nklugman@umich.edu

Walter S. Lasecki
Computer Science & Eng.
University of Michigan
wlasecki@umich.edu

Abstract

Software development is a complex task with inherently interdependent sub-components. Prior work on crowdsourcing software engineering has addressed this problem by performing an a priori decomposition of the task into well-defined microtasks that individual crowd workers can complete independently. Alternatively, ad hoc teams of experts recruited from online crowds can remotely collaborate, avoiding the up-front cost to end users of task decomposition. However, these temporary ad hoc teams can lead to high coordination costs during the session itself. In this paper, we explore the types and causes of these coordination costs for transient software teams in existing collaborative programming tools: a version control system and a real-time shared editor. Based on our findings, we suggest design elements of shared programming environments that help teams effectively self-coordinate on their task.

Author Keywords

Software development tools; Ad hoc teams; Crowdsourcing

ACM Classification Keywords

H.5.m [Information interfaces and presentation]: Misc.

Ad Hoc Crowd Programming Teams

Software development is a complicated process that frequently requires a diverse range of skills and insights. Crowdsourcing has the potential to make software production more flexible, scalable, and efficient, but this is difficult due to the inherent interdependency between subcomponents in software code. As a result, coordination costs grow significantly as team size increases [1]. In this paper, we motivate an approach to coordinating ad hoc teams in which workers can collaborate to write code remotely in real-time so that the team can complete more complex and moderately defined tasks quickly.

Crowdsourcing recruits groups of workers through an open call [24], and has been used to complete complex tasks [13, 16], and even continuous real-time tasks [15, 17]. Prior work in crowdsourced software engineering has leveraged the crowd using a number of different models [19]. For instance, Topcoder leverages a community of programmers using a competitive model where contributors participate in programming contests [14]. Latoza et al. suggest a systematic approach to decomposing a complex programming task into a set of microtasks that can be quickly solved by individual crowd-workers [18]. However, such approaches add overhead, both at the initial stage when preparing well-defined tasks, and later when results need to be integrated [23]. Ad hoc teams leave task decomposition and delegation to workers themselves, but without structured coordination or workflow design, can be inefficient [20].

Additionally, online programming assistance services that simulate remote pair-programming (synchronous) [10, 12] and services that provide programming assistance integrated into version control systems or workplace collaborative platforms (asynchronous) [4] have launched in the last few years. Codeon realizes asynchronous crowd-assisted programming through an on-demand support model, improving developer productivity by 70% over state-of-the-art tools [2, 3]. As these platforms mature, it is important that we explore rich and efficient ways to interact with crowds of programmers. Our work anticipates new assistance platforms that take advantage of the scalability of crowds while maintaining the benefits of tailored support.

This paper targets cases that developers need to hand off part of a software development task to a temporarily formed ad hoc programming team for a relatively short time span (e.g., hours, days), which will help the developer parallelize their efforts without hiring another developer long term. In this paper, we first study the issues that arise when multiple crowd workers are asked to work in an ad hoc team. We then propose a real-time shared programming environment that helps workers self-organize their collaboration.

Coordination Costs in Ad Hoc Teams

Modern programming environments support collaborative programming in various ways. Version control systems, such as `git`, are widely used in collaborative programming as programmers can work in distributed manner and synchronize easily with the main code repository. In this case, resolving merge conflicts requires additional effort and communication, often making programmers move away from collaborative programming projects, especially given the short time-span that we focus on here. Collabode [7] introduced a system that addressed the issue of breaking the collaborative build without introducing the latency and overhead of explicit version control. We conduct an experiment to further understand the coordination issues and costs that arise when groups of programmers are asked to complete a programming task without clear individual sub-goals.

At the software level, real-time shared environments [9, 21] can help mitigate much of the coordination costs between workers because: 1) the system only maintains one master copy, so individuals need not worry about code integration, and 2) the most up-to-date code is visible to everyone, meaning that workers can prevent potential conflicts and redundancies more easily before they propagate. Web-based IDE tools, such as Koding [22] and Cloud9 [11], enable users to code collaboratively in real-time. However, the needs of coordinating task decomposition and delegation are left up to users. Furthermore, allowing simultaneous access to a shared resource can cause new problems, such as corrupting someone else code. While these challenges can be addressed with in-person communication and organizational efforts (roles, responsibilities, team structures) in traditional software development teams, the inherently temporary nature of crowd teams necessitates additional tools and methods for self-coordination. To identify challenges in coordinating ad hoc teams, we conduct a user study in two widely used types of tools: version control systems (VCSs) and a shared editors, each accompanied by a call/chat (Skype).

Identifying the needs of self-coordination

User Study: Ad Hoc Programming Teams

We conducted a usability study simulating a scenario in which an end user developer hires crowd workers to form a small ad hoc team (2-3 people) and to complete a short programming task in an hour. The study had three conditions: **C1**, individual programming (1 programmer); **C2**, programming in group on a shared-code editor (2-3 programmers); and **C3**, programming in group with support of a version control system (VCS). All participants used an IDE (atom.io), and the participants in (**C2**) used a plug-in (atom-pair, https://atom.io/packages/atom-pair) that synchronizes code text, while the participants in (**C3**) used a version control system (`git`) in addition to the editor. Two collaborative groups (**C2**, **C3**) were also connected through Skype.

We recruited 12 participants (from authors' university (7) and UpWork (5)) and conducted two sessions per condition (E1-E6; refer to Table 1). Every participant had more than a year of web programming experience, and are either a freelancer or a senior undergraduate student. Each participant was asked if they were familiar with the programming concepts necessary to solve the task (regex, event handlers, and selectors). Participants were asked to complete a task in a group of two or three or independently (max time: 60 minutes). The task was to create a simple web application that takes a text content and evaluate the readability of the content by calculating various statistics (word count, letter count, five extra readability index). The task can be decomposed into a set of subtasks easily and they are dependent on one another or share common functionalities.

Participants in a group did not know each other and were asked to work on the task collaboratively with no guidance as to how to collaborate beyond using the designated tools. All participants were connected to the experimental session through the conference call and were asked to record their screen. In the end, each participant was asked to fill out a survey that has a set of open-ended questions about the collaborative programming experience. Code results submitted by the teams were evaluated based on how many test cases the program satisfied, as well as the authors' assessment of the code quality. After the experimental sessions, we analyzed the screen and voice recording to identify all of the communications between programmers. We also analyzed the effort spent coordinating the team's efforts in two different environments during the session.

Result: Shared-code Editor vs. Version Control System

It is worth noting that we did not attempt to confirm if one of the conditions outperforms any other. Rather, we observed how they collaborate, identified incidents where programmers coordinated their efforts, and collected par-

Condition	individual(C1)		shared editor(C2)		version control system(C3)	
Experiment	E1	E2	E3	E4	E5	E6
Number of Participants	1 (W)	1 (W)	2(S,S)	3(S,S,W)	2(W,W)	3(S,S,S)
Time Taken (in min)	60	60	60	58	60	60
Evaluation (max 100)	80	55	69	75	73	71.5

Table 1: We ran six experiments (E1-E6) with different conditions. For condition 2(**C2**), participants used a shared editor and for condition 3(**C3**), participants used a version control system (git). **W** indicates a crowd worker and **S** indicates a university student.

ticipant feedback from the survey. We observed two collaborative conditions that necessitated coordination effort from the team. For the version control system condition (**C3**) in which code text was not shared in real time, two groups took opposite approaches tool their collaboration. The first team (**E5**), composed of two crowd workers, split the work initially, wrote code in parallel, and merged individual code at the end. There was minimal interaction between the two programmers: it was limited to task distribution in the beginning and for code integration at the end. The consequence of two programmers working in parallel was JavaScript code in two different styles, i.e., one used regular expressions with jQuery, while the other used character-by-character comparison using arrays in pure JavaScript. While this did not hurt the correctness of the code, the style of the code in a file was not consistent which may lead to higher maintenance cost in the future.

On the other hand, the second team **(E6)** chose to communicate actively from the beginning and discussed how they could avoid merge conflicts when pushing code to the repository. They chose to create a JavaScript file per subtask, which complicated the coordination process and added the significant overhead of time (40% of total time).The first team **(E5)** spent 21% of the allotted time splitting the work into two subtasks, updating/merging their code with others, and testing the merged one. During merging the code, only one of two programmers was working in testing and validating the code. Similarly, team **(E6)** also had moments where a programmer asked others to wait and not to commit any code until they pushed the code. While two VCS groups chose different strategies for collaboration, we found they ran into the common bottleneck: task completion was deferred by configuring collaboration in the beginning and merging code at the end. The time it takes to coordinate collaboration in version control systems would have been significantly less if they were using the shared code editor.

For real-time code sharing condition(**C2**), we observed that maintaining single global “live” copy of code facilitated collaboration; this allowed participants to have access to more information, which results in more consistent code and initiates communication. They expressed the benefits of reading someone else’s code in real time; (E4-2) wrote that they “ *avoided looking into online docs for some details*” and (E4-1) noted that “*the other programmers thought of a code organization that I didn’t think of.*”). On the other hand, some people expressed that they felt “distracted” (E3-1,2) as they cannot test their code due to the incomplete code of others. This problem of being corrupted by code-in-progress in a shared editor has been addressed in [7]. However, the style of the code from (**C2**) was evaluated to be stylistically more consistent and readable than the ones from the (**C3**) group, leading to less cost for later integration and maintenance [6]. In general, we see that the advantages of using a real-time shared editor outweighed the technical difficulties in its performance and testing. Also, both groups in (**C2**) spent time in coordinating task decomposition, which potentially explains why the durations taken in the collaborative sessions are similar to the ones in the solo session (**C1**).

Result: Needs for Communication and Awareness
We discovered that the level of communication could be drastically different per group. The lack of communication can be attributed to technical issues as well as social norms (language barriers and lack of familiarity with strangers' coding styles). For example, it took 12 minutes in a session **(E5)** to split the task into two parts and the participants never communicated to each other except when merging code into the repository towards the end, resulting in the inconsistent style of the code. We found from the videos that the groups who did not communicate actively faced further issues (e.g., wasted time on redundant tasks or inconsistent code). One participant (E4-3) commented that *"it would be much more efficient if we knew each other due to better communication,"* and (E5-1) responded that the task *"should have been reviewed and discussed in depth beforehand to determine the dependency of tasks."* While the level of communication can vary depending on the different factors, the potential lack of communication necessitates nuggets of information that will help initiate and facilitate communication among programmers.

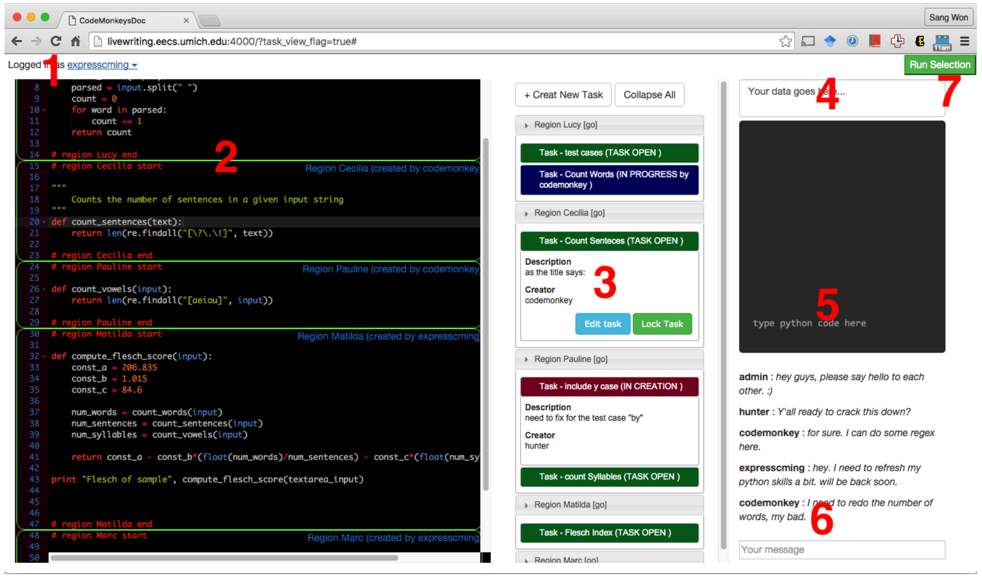
Further, participants expressed the needs of awareness in the task distribution and its progress. Participant (E4-2) commented that *"what was difficult is to understand who does what at this moment."* and (E5-2) wrote that they would like *"a system that would monitor tasks that the programmer is busy with and distribute this information to the other users."*. Various features to support simple awareness are used in shared environments [5, 8], and typically highlight edits (or cursors), which files are active, and users' connection status [21]. While such awareness features are useful for determining the spatial location of cursors, or which file is being edited, they are not sufficient to provide high-level information on the task distribution and overall progress. Simple awareness features could also mislead collaborators – e.g., the location of an inactive cursor while a programmer searched for online materials made one participant confused that it was *"difficult to determine if anybody is editing some functions in real time and decide if I can edit it"* (E4-2).

Finally, we found that early assignment of multiple subtasks to individuals can lead to a potential bottleneck that makes part of the group wait on a programmer to complete their subtasks. In two collaborative sessions (**E4, E6**), we observed that participants realized (as they wrote code) the dependencies among their subtasks, and then determined that they needed to change their assignments on-the-fly or wait for others to finish certain subtasks. The potential workaround to this problem is to assign only one task at a time so that the interdependency of sub-components that emerge later can be easily handled.

A Shared Code Editor for Ad-Hoc Teams
Based on the initial insights from the analysis of our experiments and survey results, we find that the following design elements would help reduce the coordination costs in a collaborative programming environment for ad hoc teams:

- a shared-code editor that avoids multiple versions
- displaying information on coordination that facilitates communication among team members
- self-coordination tools for programmers to flexibly complete tasks and provide progress awareness

To address these issues in communication and coordination, we are currently developing a shared code editor that facilitates self-coordination and communication (Figure 1). We introduce a subtask view that will help programmers self-organize their work below in addition to basic functionality like code synchronization, log in/out functionality, a chat interface for real-time communication, and an integrated run-time environment.



```
    parsed = input.split(" ")
    count = 0
    for word in parsed:
        count += 1
    return count

# region Lucy end
# region Cecilia start                         Region Cecilia (created by codemonkey)

"""
    Counts the number of sentences in a given input string
"""
def count_sentences(text):
    return len(re.findall("[\?\.\!]", text))

# region Cecilia end
# region Pauline start                         Region Pauline (created by codemonkey)

def count_vowels(input):
    return len(re.findall("[aeiou]", input))

# region Pauline end
# region Matilda start                         Region Matilda (created by expresscming)

def compute_flesch_score(input):
    const_a = 206.835
    const_b = 1.015
    const_c = 84.6

    num_words = count_words(input)
    num_sentences = count_sentences(input)
    num_syllables = count_vowels(input)

    return const_a - const_b*(float(num_words)/num_sentences) - const_c*(float(num_sy

print "Flesch of sample", compute_flesch_score(textarea_input)



# region Matilda end
# region Marc start                            Region Marc (created by expresscming)
```

Task - test cases (TASK OPEN)
Task - Count Words (IN PROGRESS by codemonkey)
Task - Count Senteces (TASK OPEN)
Description
as the title says:
Creator
codemonkey
Task - include y case (IN CREATION)
Description
need to fix for the test case "by"
Creator
hunter
Task - count Syllables (TASK OPEN)
Task - Flesch Index (TASK OPEN)

admin : *hey guys, please say hello to each other. :)*

hunter : *Y'all ready to crack this down?*

codemonkey : *for sure. I can do some regex here.*

expresscming : *hey. I need to refresh my python skills a bit. will be back soon.*

codemonkey : *I need to redo the number of words, my bad.*

Figure 1: a shared editor with subtask view. 1) log in 2) a shared editor with region boundaries 3) subtask-view 4) input data form 5) console output 6) chat interface 7) run button

Subtasks as a Communication Channel
The subtask-view (Figure 1-3) enumerates the list of subtasks that are created by programmers. Programmers can define a subtask and associate a region in the code editor with the subtask before writing any code. Therefore, one should declare what the subtask will be about first, before writing code. Information about subtasks, such as the title and description, are shared in real-time as they are entered. This real-time information serves as means of declaring the sub-components of the code and indicating how the overall task is decomposed into a set of subtasks up to the moment. While we apply this method to one file and the unit of subtask is a region, note that the unit can be bigger depending on the scale of a project, programming languages being used, and its programming environment.This can help programmers understand what subtasks are defined and coordinate their roles in the session by reviewing existing tasks and creating new ones. In addition, forcing programmers to enter a title and description also helps document the code. Creating subtasks is the primary way to decompose tasks, and anyone can take that initiative.

Task Distribution by Locking Subtasks
In addition to the subtask view being a declarative and communicative medium for crowd workers, locking mechanisms of subtasks help programmers be aware of the task distribution status. To be able to write code in a region of the code editor, a programmer needs to lock a subtask that is associated with the region. The locking mechanism is designed to be exclusive so that one can only lock one subtask (and thus one region) at a time. Three states of a subtask (`locked` - blue, `available` - green, or `in-creation` - red) represent what tasks are available to prevent potential conflict. Since creating a subtask is separate from locking the task, task distribution will be delayed until the moment a programmer locks the task. Assigning subtasks is done by individuals, making the process of writing code a part of the system's self-coordination mechanism.

Conclusion

We have presented initial results from a user study of ad hoc team programming to understand coordination costs in collaborative programming environments for crowd workers, and proposed a collaborative programming environment that facilitates self-coordination and communication. A set of interesting challenges remain as a future work: e.g., recruiting expert crowd workers, run-time environments for the shared editor, and code refactoring across regions.

REFERENCES

1. Frederick P Brooks. 1975. *The mythical man-month*. Vol. 1995. Addison-Wesley Reading, MA.
2. Yan Chen, Sang Won Lee, Yin Xie, YiWei Yang, Walter S Lasecki, and Steve Oney. 2017. Codeon: On-Demand Software Development Assistance. In *Proceedings of the 2017 SIGCHI Conference on Human Factors in Computing Systems*. ACM.
3. Yan Chen, Steve Oney, and Walter S Lasecki. 2016. Towards Providing On-Demand Expert Support for Software Developers. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 3192–3203.
4. Crowdbotics. 2014. (2014). `https://crowdbotics.com/` Accessed: January, 2017.
5. Paul Dourish and Victoria Bellotti. 1992. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*. ACM, 107–114.
6. Robert L Glass. 2002. *Facts and fallacies of software engineering*. Addison-Wesley Professional.
7. Max Goldman, Greg Little, and Robert C Miller. 2011. Real-time collaborative coding in a web IDE. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 155–164.
8. Carl Gutwin and Saul Greenberg. 2002. A descriptive framework of workspace awareness for real-time groupware. *Computer Supported Cooperative Work (CSCW)* 11, 3-4 (2002), 411–446.
9. Chih-Wei Ho, Somik Raha, Edward Gehringer, and Laurie Williams. 2004. Sangam: a distributed pair programming plug-in for Eclipse. In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. ACM, 73–77.
10. Codementor Inc. 2014. Code Mentor, https://codementor.io/. (2014). `https://codementor.io/` Accessed: April, 2016.
11. Cloud9 IDE Inc. 2010. Cloud9 IDE, https://c9.io. (2010). `https://c9.io` Accessed: April, 2016.
12. HackHands Inc. 2015. Hack.hands(), https://hackhands.com/. (2015). `https://hackhands.com/` Accessed: April, 2016.
13. Aniket Kittur, Boris Smus, Susheel Khamkar, and Robert E Kraut. 2011. Crowdforge: Crowdsourcing complex work. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 43–52.
14. Karim R Lakhani, David A Garvin, and Eric Lonstein. 2010. Topcoder (a): Developing software through crowdsourcing. (2010).
15. Walter S Lasecki, Christopher Homan, and Jeffrey P Bigham. 2014. Architecting real-time crowd-powered systems. *Human Computation* 1, 1 (2014).
16. Walter S Lasecki, Juho Kim, Nicholas Rafter, Onkur Sen, Jeffrey P Bigham, and Michael S Bernstein. 2015. Apparition: Crowdsourced User Interfaces That Come To Life As You Sketch Them. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 1925–1934.
17. Walter S Lasecki, Kyle I Murray, Samuel White, Robert C Miller, and Jeffrey P Bigham. 2011. Real-Time Crowd Control of Existing Interfaces. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 23–32.

18. Thomas D LaToza, W Ben Towne, Christian M Adriano, and André van der Hoek. 2014. Microtask programming: Building software with a crowd. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM, 43–54.

19. Thomas D LaToza and Andre van der Hoek. 2016. Crowdsourcing in Software Engineering: Models, Motivations, and Challenges. *Software, IEEE* 33, 1 (2016), 74–80.

20. Daniela Retelny, Sébastien Robaszkiewicz, Alexandra To, Walter S Lasecki, Jay Patel, Negar Rahmati, Tulsee Doshi, Melissa Valentine, and Michael S Bernstein. 2014. Expert crowdsourcing with flash teams. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM, 75–85.

21. Stephan Salinger, Christopher Oezbek, Karl Beecher, and Julia Schenk. 2010. Saros: an eclipse plug-in for distributed party programming. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 48–55.

22. Devrim Yasar Sinan Yasar. 2012. Koding. (2012). `https://koding.com` Accessed: April, 2016.

23. Klaas-Jan Stol and Brian Fitzgerald. 2014. Two's company, three's a crowd: a case study of crowdsourcing software development. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 187–198.

24. James Surowiecki. 2005. *The wisdom of crowds*. Anchor.