Exploring Coordination Models for Ad Hoc Programming Teams

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

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

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

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

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

Walter S. Lasecki
Computer Science and Eng.
and School of Information
University of Michigan
wlasecki@umich.edu

Abstract

Software development is a complex task with inherently interdependent sub-components. Prior work on crowdsourcing software engineering address this problem by performing a prior decomposition of the task into well-defined microtasks that individual crowd workers can complete. Alternatively, recruiting ad hoc teams of crowd programmers to (remotely) collaborate on completing a programming task avoids the up-front cost to end users, but can lead to high coordination costs. In this paper, we explore the types and causes of these coordination costs in remote ad hoc teams that coordinate using either a version control system (VCS) or synchronous editor, and compare overall performance to that of independent workers. Based on our findings, we propose a shared programming environment designed to help teams efficiently coordinate by integrating features from both version control and synchronous coordination.

Author Keywords

Software development tools; Ad hoc teams; Crowdsourcing

ACM Classification Keywords

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

Software development is a difficult process that frequently requires a diverse range of skills and insights. Crowdsourcing software engineering has the potential to make software production more flexible, scalable, and efficient. However, this is difficult due to inherent interdependencies in complex projects, which means coordination costs grow significantly with larger teams [2].

Existing work in crowdsourced software engineering has successfully leveraged the crowd using a number of different models [13]. For instance, Topcoder leverages a community of programmers using a competitive model where contributors participate in programming contests [10]. 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 [12]. However, such approaches add overhead, both at the initial well-defined task preparation stage and later at the outcome integration stage [18]. Ad hoc teams leave task decomposition to workers themselves, but without structured coordination and workflow design, can be inefficient [16].

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.

Related Work

Pair programming [19], where two software engineers work together in real-time, has been shown to produce better quality output with reasonable overhead in effort [4, 15]. However, crowdsourcing platforms make developers available to work in parallel, similar to side-by-side programming, which is more efficient than pair programing [14].

At the software level, real-time web-based code editors [7, 17] 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 merging code, and 2) the most up-to-date code is visible to everyone so that workers can coordinate conflicts and redundancies before they propagate. However, allowing simultaneous access to a shared resource can cause new conflicts. Most importantly, workers are still left with the issue of structuring collaboration, with the aid of an additional communication channel (e.g., voice call).

Outside of a programming context, there have been approaches to encouraging structured collaboration in crowdsourcing systems. CrowdForge presents a general purpose framework for accomplishing complex and interdependent tasks via decomposition into discrete “map” tasks, which are carried out by more than one worker [9]. Kim et al. explore structured collaboration in creative writing, finding that self-assignment of roles helps participants complement each other’s skills, leading to better collaborations [8]. André et al. demonstrate that sequential work becomes more effective than simultaneous work as the size of a collaborative group increases, but can be overcome through role assignments [1]. Apparition avoids conflicts on a shared canvas through a write-lock mechanism [11].

Given such related work, we conduct a user study to better understand the issues that arise in real-time collaboration when workers use either a shared editor or a version control system (VCS). We then propose a shared-editor system that supports worker-driven task decomposition during a programming task.

Coordination Costs in Ad Hoc Teams

We conducted an experiment to understand the coordination issues and costs that arise when groups of programmers are asked to complete a task with no clear individual goals. Our experimental design simulates a scenario where an end user hires workers from a crowd platform (e.g., UpWork) to form a small ad hoc team (2-5 people) to complete a programming task.

Experiment	E1	E2	E3	E4	E5	E6
Condition	Individual(C1)	Individual(C1)	Shared Code(C2)	Shared Code(C2)	VCS(C3)	VCS(C3)
Number of Participants	1	1	2	3	2	3
Time Taken (in min)	60	60	60	53	60	60
Task 1 Score (max 100)	80	55	69	75	73	71.5
Task 2 Score (max 100)	80	70	(80, 80)	(72, 64, 60)	(100, 40)	(20, 100, 20)

Table 1: We ran six experiments (E1-E6) with different conditions (e.g., whether the programmers worked individually or used Github as the VCS) and show the corresponding performance metrics. We assess scores based on a rubric that considers functionality and is weighted by difficulty and importance.

Property	Score
Number of Letters	10
Number of Words	15
Number of Sentences	15
Number of Syllables	20
Flesch reading ease	10
Coleman-Liau Index	10
Display Event Handler	20
Total	100

Table 2: Task 1 (Readability Stats) Rubric.

Experimental Setup
We compare three conditions: **C1**, individual programming (1 programmer); **C2**, programming in group on a shared-code editor (3 programmers); and **C3**, programming with support of a version control system(or VCS) to share code among workers (3 programmers). In terms of tools: **C1** uses a traditional offline code editor, **C2** uses an existing shared-code package of the editor (atom-pair, https://atom.io/packages/atom-pair) that synchronizes code text in real-time using a cloud service, and **C3** uses Github, a widely-used VCS that lets workers write code separately and then merge results by "committing" the code. Two collaborative groups are assisted with the conference call (voice and chat enabled) and all participants were asked to use a same editor (Atom.Io).

We recruited fourteen participants (from authors' university (7) and UpWork (7), all male) and conducted two experiments per condition (E1-E6; refer to Table 1). Two Upwork workers left the session during the experiment before starting the tasks. Participants completed two tasks: The first task asks teams to calculate statistics on web page content without any collaboration instructions (max time: 60 minutes), so this task helps delineate issues the ad hoc team faced while coding collaboratively. The second task asks each individual to write a phone number format checker (max time: 10 minutes). This task helps measure individual proficiency. At the end, participants were asked to fill out a survey regarding the experience. We use the submitted code and participants' screen recordings for analysis and identified performance bottlenecks. In turn, these roadblocks inform design of our on-going work on a new team coding editor.

Result: Code Quality
We assess results with a rubric (Table 2) that considers functionality and weights scores based on task difficulty and importance. For task two, we use four test cases (Table 3) to see if the format checker yields false positives or false negatives. Our current results suggest that ad hoc teams do not outperform individuals. We believe one potential reason for this is due to the real-time shared editor's lack of debugging functionality. Based on screen recordings, we see that this adversely impacted code quality. We posit that more debugging support will improve quality; nevertheless, here, we focus primarily on the issues that ad hoc groups encounter when coordinating tasks.

Result: Shared-code Editor vs. Version Control
For the **C3** groups (using VCS), we discovered a common costs of using such system, even though each group took different approaches. The first team (E5), composed of two workers from Upwork (after one worker disconnected), coordinated with the following approach: split the work initially, write code in parallel, and merge individual code at the end. Naturally, the final code contained two different styles of code – e.g., one used regular expressions with jQuery, while the other used character comparison in pure Javascript. This team spent 21% of the allotted time not to generate/improve code but just to update/merge their code with others and testing the merged one. On the other hand, the second team (E6) chose to communicate actively from

Property	Score
Hit	20
No False Positives	20
No False Negatives	20
Correct Rejection	20
Event Handler	20
Total	100

Table 3: Task 2 (Phone Number format Checker) Rubric. For example, false positive means that the checker says "123-456-7z90" is a valid output, even though the string is not a valid number.

the beginning and discussed how they can avoid pushing conflicted code to the repository. They chose to create a javascript file per subtask, which added about 40% overhead.

While both VCS groups chose different strategies for collaboration, we found they ran into the same bottleneck: task completion was blocked by each others' progress. Programmers were asked or needed to wait until some coordinating tasks were done and updated in the repository.

In contrast, the shared-code editor condition (C2) posed a different challenge to the participants. Besides technical problems with the tool itself (latency and synchronization bugs), the participants expressed their difficulty in testing their code. For instance, if another programmer opens a comment block, code cannot be tested unless the block is closed. This problem of being corrupted by code-in-progress in a shared-editor has been explored by Goldman et al. [6].

Despite difficulties, real-time code sharing appears to facilitate collaboration indirectly by maintaining a global "live" copy of code; this allowed participants to have access to more information, which potentially improves code quality and overall efficiency (participant E4-2 wrote that they "avoided looking into online docs for some details"). Participants favored the shared-code editor; indeed, four out of five participants responded that being able to see others code in real time helped with their own approach (although two respondents also said it was distracting due to the aforementioned technical/debugging problems). Lastly, although we did not include the style of the code in the grading, the code from the shared-editor groups outperformed the code from the CVS group, leading to less cost for later integration and maintenance [5].

When compared with the VCS case above, testing of the program in the shared editor becomes the main challenge (but everyone has access to the real-time evolving global copy), whereas the primary difficulty with VCS is programmer dependency (but testing can be done locally). Therefore, another design objective is to have a shared-view of the code, but still allow for localized testing.

Result: Communication and Asynchronous Awareness
In all group experiments, participants used a conference call system to self-coordinate and to split the task into subtasks that can be done in parallel. However, what we discovered is that the level of communication can be drastically different per group: those groups that actively communicated were able to coordinate tasks more easily; as expected, those groups that did not communicate as frequently faced further issues. We discovered that the lack of communication can be attributed to technical issues (frequent disconnection of the audio system), as well as social issues (language barriers and lack of familiarity with strangers' coding styles).

In fact, the need for awareness was a major concern (as shown by Dourish and Bellotti [3]): one participant (E4-3) commented that *"it would be much more efficient if we knew each other due to better communication,"* while the other (E4-2) responded that *"calibrating on what everyone's experience levels were and strengths/weaknesses would have been good for us to do before starting. (...) because programming with people you don't know is somehow kind of unsettling."* Furthermore, awareness in terms of which subtask is being completed by which programmer is also of crucial concern. Participant E5-2 wrote that a better implementation of the code editor *"[...] is a system that would monitor tasks that the programmer is busy with and distribute this information to the other users."*

Various features to support such awareness are used in shared-environments, typically highlighting edits (or cursor) and an active file [17]. However, from the experiment, we discovered that awareness can be lost in the shared code in real-time. First, such awareness feature may have incorrect information. During the study we noticed that all the participants spend a significant amount of time beyond editing (that includes searching materials on a web browser or reviewing code in a viewport that the cursor is not in it); this means that the cursor is currently inactive. This may mislead a programmer to believe that a certain part of the code is available to edit, but in reality, the cursor is inactive because that programmer is looking for material on the web. On the flipside, a misplaced and inactive cursor may lead others to think that the owner of the cursor is going to work on it soon—although the code region is truly inactive. Second, such awareness features only provides one bit of information (i.e., whether the code region is actively being developed on). It is not clear who wrote a certain part of the code in the past, especially in the team of more than two people.

Another important finding from the experiment is that early assignment of subtasks to individuals can lead to a bottleneck that makes part of the group wait on a programmer to complete their subtask. A potential workaround is to have participants take one task at a time so that the coordination with work distribution can be switched on the fly based on the availability of individual programmers.

All in all, lack of communication and the pitfalls in awareness can adversely impact task completion, which in turn degrades code quality. We believe these challenges will scale with team size.

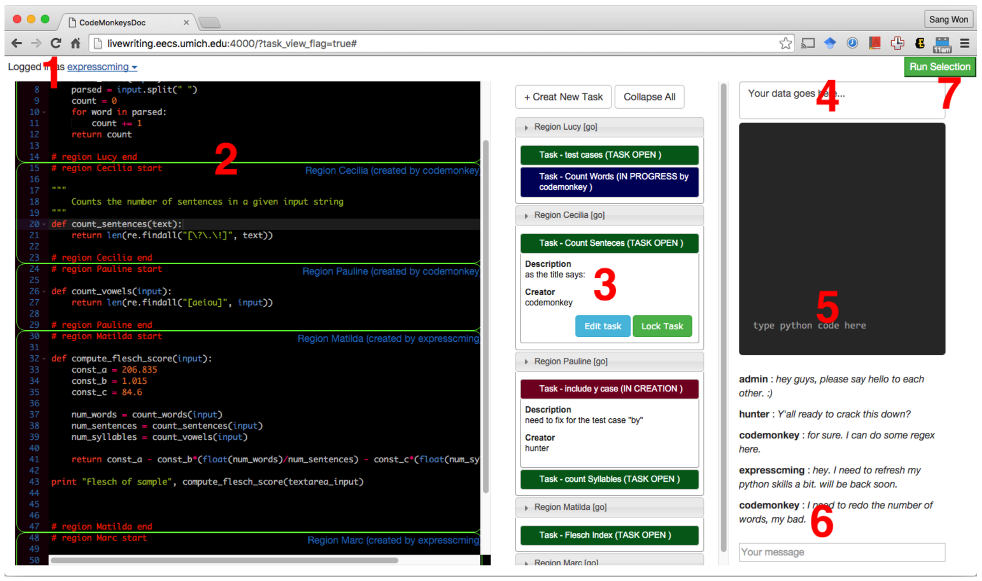
On-Going Work: A New Team Code Editor

Based on our experiments, we discovered that it is advantageous to use a shared-code editor over using only a VCS. However, we discovered issues that arose in the shared code editor setup: 1) difficulty in testing/debugging due to work-in-progress; 2) potential lack of communication; and, 3) asynchronous awareness. To address these issues, our ongoing work is exploring a shared code editor with a subtask creation and a subtask locking mechanism integrated in the system. These design choices help solve issues (2) and (3). We are developing a new web-based programming environment that does include code-sharing among programmers connected in a collaborative session. The editor includes code synchronization, log in/out functionality, a chat interface for real-time communication, and a runtime environment (currently Python).

Subtask creation as a communication channel

The subtask-view (Figure 1-3) allows programmers to create subtasks and define a region in the code associated with the subtask. By default, one cannot write code in the editor. Instead, one creates a locking subtask from the list inside the region that is associated with the task. The region will be automatically expand as the code size grows, and can help future automated system better understand task decomposition. When creating a subtask, a programmer must enter information about the task, such as title and description (Refer to Figure 1).

The process of subtask creation will be shared in real time in keystroke level so that one can immediately see what kinds of subtasks are in creation by others. Other programmers can get live information about: 1) how the programming task given is decomposed to a set of subtasks up to the moment, and 2) what kinds of subtask are not yet declared. Creating a subtask serve as a mean of displaying


CodeMonkeysDoc
livewriting.eecs.umich.edu:4000/?task_view_flag=true#
Sang Won
Logged in as expresscming
Run Selection
1
2
3
4
5
6
7

```
    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)
```

+ Creat New Task | Collapse All

Region Lucy [go]
- Task - test cases (TASK OPEN)
- Task - Count Words (IN PROGRESS by codemonkey)

Region Cecilia [go]
- Task - Count Senteces (TASK OPEN)
  - Description: as the title says:
  - Creator: codemonkey
  - Edit task | Lock Task

Region Pauline [go]
- Task - include y case (IN CREATION)
  - Description: need to fix for the test case "by"
  - Creator: hunter
- Task - count Syllables (TASK OPEN)

Region Matilda [go]
- Task - Flesch Index (TASK OPEN)

Region Marc [go]

Your data goes here...

type python code here

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.*

Your message

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

the process of task decomposition to collaborators in real time without initiating verbal/chat communication, which is hard to keep track of when scaled. We should note that, by no means do we imagine programmers will be able to collaborate without traditional means of communication if they use the subtask view. Rather, we expect that programmers can use it for task decomposition with less pressure and the created subtasks will help initiate communication in traditional senses.

Subtask for Asynchronous Awareness
In addition to the subtask being a declarative and communicative medium for others, subtasks created in the list will support asynchronous awareness. The task is sorted by the region and the time of creation with its status visualized so that one can keep track of what kinds of the task has been in creation, available, or in-progress. The status display of a locked task will be an indicator that represent that the subtask is currently assigned to a programmer and in progress so that the programmer who locks the task does not have to fear the conflict and freely be inactive for other activities (e.g. searching syntax, copy and paste from other regions). Subtask creation and locking the task are not necessarily done by the same programmer. One can only work on one task at a time; any task in the list is for any available programmer to lock and start working on. A region can be associated with multiple subtasks so that one can define a series of subtasks on the same region (e.g., separate tasks for coding, testing, and debugging). To prevent conflicts with others even before starting to write code or searching related information on a web browser, one can lock a task which declares that the worker is working on a certain task.

Conclusion
We have presented initial results from a user study of ad hoc team programming, and we presented on-going work on a system that integrates task-decomposition functionality into a shared-code editor. Subtask can be used to keep track of dynamically evolving code to augment existing communication channels (voice/chat), and help future systems learn from worker task decomposition over time.

There remain interesting research questions of using this subtask creation/locking feature, such as: 1) how the overhead of creating subtasks can be overcome, 2) how programmers can freely test and debug their synchronously-shared code, 3) how the quality of the code from a shared-editor can be scaled, and 4) how the consistency among subtasks can be maintained in larger projects. We believe there exist exciting opportunities to generalize the work we have started here to support more flexible, scalable collaborative software engineering through online crowds. We look forward to the chance to discuss this with CHI attendees.

References

[1] Paul André, Robert E Kraut, and Aniket Kittur. 2014. Effects of simultaneous and sequential work structures on distributed collaborative interdependent tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 139–148.

[2] Frederick P Brooks. 1975. *The mythical man-month*. Vol. 1995. Addison-Wesley Reading, MA.

[3] 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.

[4] Tore Dyba, Erik Arisholm, Dag Sjoberg, Jo E Hannay, Forrest Shull, and others. 2007. Are two heads better than one? On the effectiveness of pair programming. *Software, IEEE* 24, 6 (2007), 12–15.

[5] Robert L Glass. 2002. *Facts and fallacies of software engineering*. Addison-Wesley Professional.

[6] 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.

[7] 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.

[8] Aniket Kittur, Bryant Lee, and Robert E Kraut. 2009. Coordination in collective intelligence: the role of team structure and task interdependence. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1495–1504.

[9] 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.

[10] Karim R Lakhani, David A Garvin, and Eric Lonstein. 2010. Topcoder (a): Developing software through crowdsourcing. (2010).

[11] 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.

[12] 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.

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

[14] Jerzy R Nawrocki, Michał Jasiński, Łukasz Olek, and Barbara Lange. 2005. Pair programming vs. side-by-side programming. In *Software process improvement*. Springer, 28–38.

[15] John T Nosek. 1998. The case for collaborative programming. *Commun. ACM* 41, 3 (1998), 105–108.

[16] 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.

[17] 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.

[18] 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.

[19] Laurie Williams and Robert Kessler. 2002. *Pair programming illuminated*. Addison-Wesley Longman Publishing Co., Inc.