On the Benefits of Providing Versioning Support for End Users: An Empirical Study

SANDEEP K. KUTTAL, ANITA SARMA, and GREGG ROTHERMEL,
University of Nebraska-Lincoln

End users with little formal programming background are creating software in many different forms, including spreadsheets, web macros, and web mashups. Web mashups are particularly popular because they are relatively easy to create, and because many programming environments that support their creation are available. These programming environments, however, provide no support for tracking versions or provenance of mashups. We believe that versioning support can help end users create, understand, and debug mashups. To investigate this belief, we have added versioning support to a popular wire-oriented mashup environment, Yahoo! Pipes. Our enhanced environment, which we call "Pipes Plumber," automatically retains versions of pipes and provides an interface with which pipe programmers can browse histories of pipes and retrieve specific versions. We have conducted two studies of this environment: an exploratory study and a larger controlled experiment. Our results provide evidence that versioning helps pipe programmers create and debug mashups. Subsequent qualitative results provide further insights into the barriers faced by pipe programmers, the support for reuse provided by our approach, and the support for debugging provided.

Categories and Subject Descriptors: D2.9 [**Software Engineering**]: Management—*Software configuration management*; D2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; H1.2 [**Models and Principles**]: User/MachineSystems—*Human factors; human information processing*; H5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Evaluation / methodology*

General Terms: Human Factors

Additional Key Words and Phrases: End-user software engineering, versioning, Mashups, Yahoo! Pipes, reuse, debugging, programming barriers

ACM Reference Format:
Sandeep K. Kuttal, Anita Sarma, and Gregg Rothermel. 2014. On the benefits of providing versioning support for end users: An empirical study. ACM Trans. Comput.-Hum. Interact. 21, 2, Article 9 (February 2014), 43 pages.
DOI: http://dx.doi.org/10.1145/2560016

1. INTRODUCTION

End-user programming has become a widespread phenomenon. Large numbers of end users (i.e., nonprofessional programmers) create software applications for their own needs by using a variety of programming tools and environments such as spreadsheets, Scratch, Labview, web macros, and web mashup environments [Scaffidi et al. 2005]. While these domains support programming (e.g., macros in spreadsheets, linking code blocks in Scratch), they lack basic software engineering support such as debugging

This work was supported in part by the AFOSR through grant FA9550-10-1-0406, and by the NSF through grants IIS1110916 and IIS1314365 and to the University of Nebraska-Lincoln.
Authors' addresses: Sandeep K. Kuttal, 103A Avery Hall, Computer Science and Engineering Department, University of Nebraska-Lincoln, Lincoln, NE 68588; Anita Sarma and Gregg Rothermel, 362 Avery Hall, Computer Science and Engineering Department, University of Nebraska-Lincoln, Lincoln, NE 68588.

DOI: http://dx.doi.org/10.1145/2560016

help, testing support, or variation management—support commonly available in professional development environments. Researchers in end-user programming focus on using software engineering principles and techniques to provide such support, with the aim of helping end users in their efforts [Ko et al. 2011].

One particular software engineering principle relates to the need to support product families (version support) and has been widely acknowledged in the professional software development community as essential for code understanding, change traceability, debugging, and maintenance [Tichy 1985]. Version support allows developers to browse past and alternative versions of a resource, determine how these versions differ from one another, and choose a particular version or revert back to a previous version.

In a domain study of experts (scientists), Jones and Scaffidi [2011] observed that there is a need for version control to enhance the maintainability and reuse of visual, domain-specific languages. In end-user programming domains, however, support for versioning is typically unavailable.

We posit that end users are likely to benefit from versioning support because they often tend to learn from examples [Lieberman et al. 2006] that they find in repositories. They also tend to "debug their programs into existence" [Rosson and Carroll 1993]; that is, they investigate different alternative strategies and backtrack their changes to arrive at a solution. Moreover, end users tend to opportunistically create their programs [Brandt et al. 2009], exploring different aspects of the programs and solutions on an as-needed basis. Versioning support can help users in their explorations, including explorations through versions in the repository as well as explorations among their own changes.

In this article, we investigate whether and how versioning support can help end-user programmers in their program creation and debugging tasks. We focus on web development and web mashups. Web mashups are applications that combine data, functionality and interface elements from two or more sources to create new services. Mashups are popular among end users because of their role in three primary trends involving the Web 2.0 paradigm, where end users (1) create dynamic content for the web, (2) build situational software applications [Huang et al. 2008], and (3) build on and share their applications through publicly hosted repositories [Jones and Churchill 2009].

Users programming mashups do not need to write scripts or programs; instead, they can take advantage of visual, black-box–oriented programming environments. Examples of such environments include IBM mashup maker,[1] JackBe,[2] Deri pipes,[3] Apatar,[4] xfruit,[5] and Yahoo! Pipes.[6] Among these programming environments, Yahoo! Pipes has been one of the most popular, and has drawn a large community of users. Yahoo! Pipes is a commercial mashup programming environment that allows users to aggregate and "mashup" content from around the web. In its first year (February 2007 through 2008), more than 90,000 developers created individual pipes on the Yahoo! Pipes platform, and pipes were executed over 5,000,000 times each day [Jones and Churchill 2009].

Mashup programming environments provide central repositories to end users where they can execute and store their mashups. However, these environments do not provide facilities by which users can keep track of the versions or provenance of the mashups they create. For example, in a study of the Yahoo! Pipes repository, it was found that

[1]IBM Mashup Maker: http://www.ibm.com/software/info/mashup-center/.
[2]Jackbe: http://www.jackbe.com/.
[3]Deri Pipes: http://pipes.deri.org/.
[4]Apatar: http://apatar.com/.
[5]xfruit: http://xfruit.com.
[6]Yahoo! Pipes: http://pipes.yahoo.com/pipes/.

43% of pipes submitted are just variations of previously submitted pipes, indicating that authors may be using the public repository as a private repository to manually store versions of their pipes containing incremental changes [Stolee et al. 2011].

Furthermore, as noted earlier, backtracking and investigating alternative scenarios are an integral part of debugging into existence—a programming paradigm that is popular with end users. Most mashup programming environments, including Yahoo! Pipes, however, cannot represent alternative exploration paths as branching histories, forcing users to rely on memory to compare scenarios. In a study of end-user mashup programmers, it was observed that all participants backtracked multiple times while creating mashups, either because they had alternative ideas or wished to return to some previous successful state [Cao et al. 2010b]. The authors also observed that participants spent 76.3% of their time in debugging alone while developing mashups [Cao et al. 2010a].

Motivated by the studies just described, we are investigating the use of versioning in mashup programming environments. We conjecture that the addition of versioning support to these programming environments will be useful in several ways, including (1) helping mashup programmers create and reuse mashups, (2) helping mashup programmers understand the evolution of mashups, (3) helping mashup programmers backtrack to successful states and explore alternative ideas, and (4) helping mashup programmers debug mashups.

To investigate this conjecture, we have added versioning support to Yahoo! Pipes. The primary reasons for selecting Yahoo! Pipes include its popularity, the fact that it is available for free, the availability of a large repository of pipes [Jones and Churchill 2009], and the fact that its data can be captured and manipulated by external systems. Our extension to Yahoo! Pipes, which we call "Pipes Plumber," keeps version histories for mashups automatically. This allows users to utilize the advantages of versioning without needing to be aware of the underlying functions known to professional programmers such as check-in, check-out, and so forth.

To explore the potential cost and benefits of our versioning support, we conducted two user studies. Our first study [Kuttal et al. 2011b], an exploratory, think-aloud study, involved a version of the Yahoo! Pipes environment augmented with basic configuration management functionality and a simple interface. This study of nine participants (primarily computer science students) showed that the versioning support helped them create mashups more efficiently.

Subsequently, we extended our versioning support by providing additional user interface assistance intended to help users debug faulty mashups. We then conducted a controlled experiment involving participants who do and do not have formal programming training and experience, studying questions related to the creation, and debugging of pipes [Kuttal et al. 2011a]. Our experiment results confirm that both treatment groups of participants can create pipes more effectively and efficiently with the aid of versioning support, and they can also debug pipes more effectively.

This article presents both of the foregoing studies and then augments this presentation with an extensive qualitative analysis of our data. Our results suggest that users had difficulty comprehending the state of pipes and their feedback (related to understanding barriers) and how to correctly use the modules provided by Yahoo! Pipes (related to use barriers). The availability of a history of evolution and the state of the program in past versions helped ease these barriers. Furthermore, when users knew that their changes would not be lost (because of automatic versioning), they were less risk averse and explored more alternative ideas. Another advantage of versioning was the fact that when debugging, users knew which strategies they had already followed and did not redo their failed strategies. Finally, versioning helped participants who had and did not have formal programming experience, and we did not find any significant differences between these two types of users.

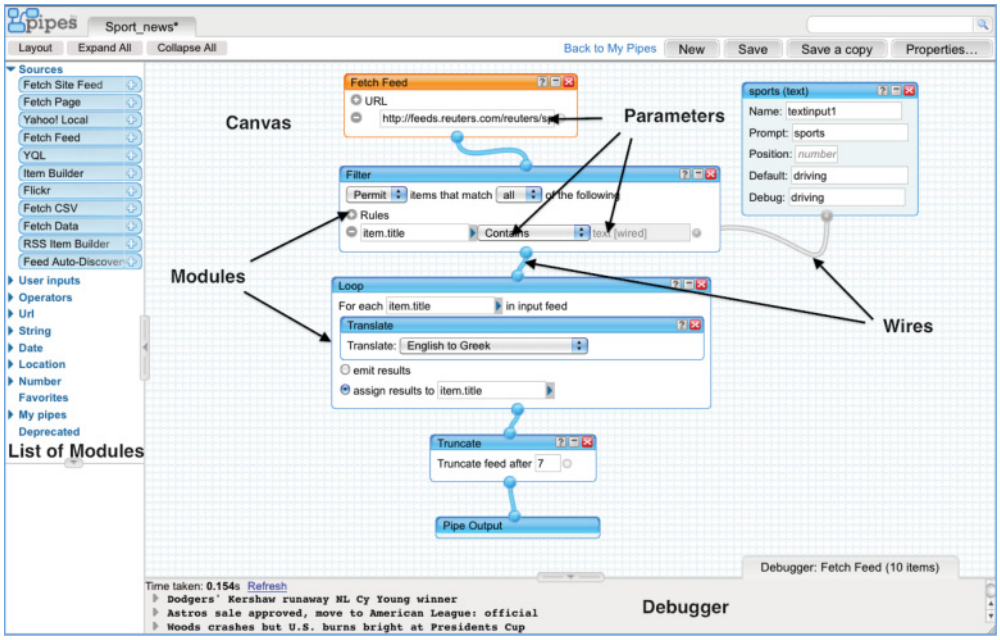

Fig. 1. Yahoo! Pipes interface.

The remainder of this article is organized as follows: Section 2 provides background information. Section 3 describes our Pipes Plumber extension to Yahoo! Pipes and its interface. Section 4 provides details on the setup and results observed in both studies. Section 5 provides our qualitative analysis of the two studies. Section 7 discusses related work. Section 8 concludes and discusses future work.

2. YAHOO! PIPES

Yahoo! Pipes[7] is one of the most popular mashup creation environments available and is used both by professional and end-user programmers. Yahoo! Pipes is a web-based visual programming environment introduced by Yahoo! in 2007 with the intent of enabling its users to "rewire the web." As a visual programming environment, Yahoo! Pipes is well suited to representing the solutions to dataflow based processing problems [Jones and Churchill 2009]. Yahoo! Pipes "programs" combine simple commands together such that the output of one acts as the input for the other. The Yahoo! Pipes engine also facilitates the wiring of modules together and the transfer of data between them.

Figure 1 shows the interface of the Yahoo! Pipes environment and the various components of that interface. The pipe displayed in the figure takes an RSS feed from Reuter's News as input and then filters the news based on input parameters supplied by the user (by default, sports). It then converts the titles of all the news feeds from English to Greek, and displays only the first seven results. The Yahoo! Pipes environment consists of three major components: canvas, library (list of modules), and debugger. The canvas is the central area where a user creates a pipe. The library is located to the left of the pipe editor and consists of various modules that are categorized according to functionality. Users drag modules from the library and place them on the

[7]Yahoo! Pipes: http://pipes.yahoo.com/pipes/.

Table I. Comparison of Features of Existing CM Systems to Features Available in Pipes Plumber

Features	CM Systems	Our System
Versioning unit	File level	Pipe level
Differencing	Text level	Module level
Deltas	Add, delete, modify	Add and delete
Versions created	On commit	On save/cloning of pipe
Browsing	Undo or select version	Undo, redo or select from list view
Tag	Baseline/tags	Run/tags
Merge	Implemented	Future work

canvas, then connect them to other modules as needed. The debugger helps users check the runtime output of specific modules including the final output module (in Figure 1 the debugger window displays the output from the module `Fetch Feed`).

Pipes programmers create their pipes using the visual interface on the client side. When they save a pipe, it is encoded in JSON format and sent to the Yahoo! Pipes server, which is where all pipes are saved and executed. Programmers may also "clone" pipes that are available in the repository, in order to reuse them in new contexts.

Input to a pipe can be HTML, XML, JSON, RDF, RSS feeds, as well as many other formats. Similarly, pipe output can be RSS, JSON, KML, and other formats. The inputs and outputs between modules are primarily RSS feed items. RSS feed items consist of parameters and descriptions. Yahoo! Pipes modules provide manipulation actions that can be executed on these RSS feed parameters. In addition to items, Yahoo! Pipes also allows datatypes like *url, location, text, number*, and *date-time* to be defined by users.

3. PIPES PLUMBER

We now discuss the approach we use to provide versioning capabilities for Yahoo! Pipes, including the features we provide, our system architecture, and the interface used.

3.1. Versioning Features

In the world of professional software development, versioning support is provided by *Configuration Management* (CM) systems. Most CM systems built for professionals (e.g., CVS, SVN, Git) require them to learn specific commands and concepts. Because end users do not have any formal training with these systems or the associated concepts, using these systems imposes additional cognitive load on them. Moreover, end users tend to program opportunistically and debug their programs into existence; hence, they iteratively backtrack and investigate alternative scenarios. Most mashup programming environments, however, including Yahoo! Pipes, do not represent prior versions or alternative exploration paths as branching histories, which forces users to rely on memory to compare scenarios. This motivated us to attempt to create a versioning system that could automatically retain version histories for mashups, allowing users to utilize the advantages of versioning without needing to understand the mechanics of underlying versioning operations.

To achieve versioning support in Yahoo! Pipes, we considered each of the versioning features available in professional CM systems and created appropriate analogous features. Table I summarizes the results, which we elaborate on next.

—In most CM systems, the *versioning unit* is a file. There is no concept of files in Yahoo! Pipes; the smallest executable unit is a pipe which is checked in (saved) or checked out (viewed) by pipe programmers. Thus, we chose to use individual pipes as our versioning unit.
—In CM systems, the *differences between two versions* of files are calculated at the text level. Pipes, in contrast, consist of modules and wires. Pipes programmers "program"

by adding or deleting modules. Thus, we chose to calculate the differences between pipes at the module level.
—In CM systems, because differencing is done at the text level, *deltas* for a file can be lines of code added, deleted, or modified. As mentioned before, we support differencing of pipes at the module level. This leads to a clear choice of treating modules (added or deleted) as deltas.[8]
—In CM systems, a *version* is created by a professional developer on each commit, which he or she deliberately performs on major changes. Our objective is to integrate versioning into the Yahoo! Pipes environment, while requiring little effort on the part of end users to learn about versioning features. End users also purposefully save their pipes to incorporate changes. Thus, we automatically create versions when users save or clone their pipes so that they need not apply extra effort to create versions.
—CM systems allow developers to browse different versions of a file by using commands to select some specific version. Such commands may increase the cognitive load for end users; thus, we enable browsing using customized widgets. With these widgets, users can perform various activities like undo, selection of a version (as in CM systems) or redo (an additional feature).
—Typically, CM systems allow developers to tag the development tree, by marking significant versions as baselines for retrieval and deployment. In the Yahoo! Pipes environment, a user who believes that his or her pipe is complete will run the pipe to check the results. We use this to facilitate the automatic creation of baselines. When a user runs a pipe, we tag that version as a baseline. In addition to baselines, in our environment, we find other forms of tags of systems useful, such as the number of results returned and the success or failure of the run. We also tag the user's tested versions of pipes for debugging.
—CM systems provide merge facilities to let programmers merge versions; this facilitates collaboration among different professionals. We do not yet support merging, but this capability could be added to our system in the future.

3.2. System Architecture

To implement versioning support, rather than create an add-on for a specific browser, we decided to create a proxy wrapper. This proxy wrapper intercepts the JSON code (pipe) that is transmitted between the user's client running in their browser and the Yahoo! Pipes server. This allows "Pipes Plumber" to be operational for most web browsers (including Internet Explorer, Chrome, Firefox, and Safari).

Figure 2 presents our system architecture. We use a proxy server (Squid 3.1.4[9]) to manage communications between the client (web browser) and the Yahoo! Pipes server. Using the Internet Content Adaptation Protocol (ICAP[10]), a proxy wrapper intercepts the request and response messages exchanged between a client and the Yahoo! Pipes server. When the user utilizes the User Interface (UI) of Yahoo! Pipes, the response related to the UI is redirected to the proxy wrapper. The proxy wrapper modifies the response messages it receives by inserting "widgets" related to versioning and debugging into the UI before delivering the message to the client.

Our proxy wrapper intercepts user events (e.g., save or run) and message contents and based on these, creates and stores versions in a MySQL database, which serves

[8]We do record modifications as we save versions in the repository; however, the listing of modules that we present to the user does not contain metadata related to such modifications.
[9]Squid: http://www.squid-cache.org/Versions/.
[10]ICAP: http://www.icap-forum.org/.

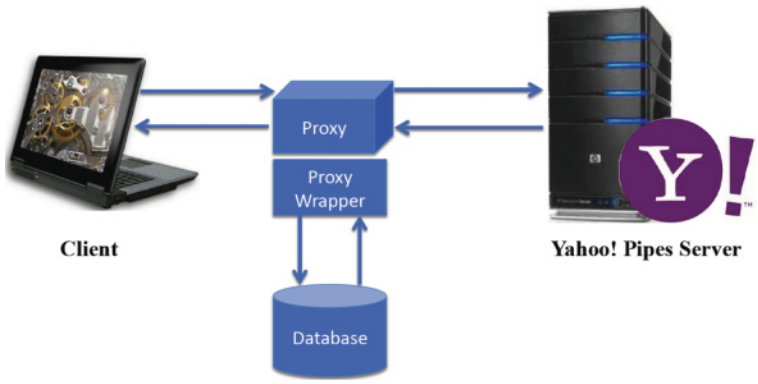

Fig. 2. Versioning support architecture.

as our versioning repository.[11] When a user saves a pipe, a new version of the pipe is created in the repository. Versions are created in chronological order (e.g., V_1, V_2, $V_3, \ldots, V_n$) in the sequence of saves. Each version is composed of the set of modules added to the canvas by the user. Each version also keeps track of its parent. When the user requests a particular version of a pipe from the Pipes Plumber interface, the requested version is selected from the repository. Hence, each version of a pipe can be viewed, edited, or run using the Pipes Plumber interface.

To illustrate the process, consider a user-based scenario. Suppose Sally wants to create a pipe using the Yahoo! Pipes environment, but she also wants the benefits of versioning. She connects with our proxy server and sends a request for the Yahoo! Pipes interface to the Yahoo! Pipes server. When the response comes back from the server, it comes through the proxy server where the proxy wrapper adds our widgets to the Yahoo! Pipes interface. Hence, Sally views the Pipes Plumber interface and is ready to use our versioning support in the Yahoo! Pipes environment.

Suppose Sally begins to create a pipe to search for movies played in theaters near a specific location. Once her task is done, she saves the pipe. As soon as she saves her pipe a save message (POST) is sent from the client machine to the Yahoo! Pipes server. Our proxy wrapper intercepts this message and saves the JSON contents from the message body as a version in the central repository. Hence, the first save of Sally's pipe creates version V_1 in the central repository. Each time Sally adds additional functionalities to the pipe (such as checking reviews or displaying posters of a movie), and later saves or clones her pipe, corresponding versions are saved in the central repository, in chronological order. (Note: the Yahoo! Pipes repository retains only the latest contents of the pipe; earlier versions are all saved by our system.)

Given the foregoing, if Sally wishes to retrieve a previous version V_2 of a pipe, she can achieve this using the Pipes Plumber interface. She selects version V_2, and this version is drawn from the local repository and sent to the Yahoo! Pipes server, where she can access or execute it.

3.3. Interface

The UI of Pipes Plumber adds four widgets to the Yahoo! Pipes client interface (see Figure 3). These widgets are (1) Undo, (2) Redo, (3) Tested, and (4) History of Pipe. The first two widgets, Undo and Redo, are buttons that allow users to browse between

[11]This approach provides an initial prototype sufficient for use in studying versioning. The database could easily be replaced, however, by an actual versioning system such as `git`; this would require only that the

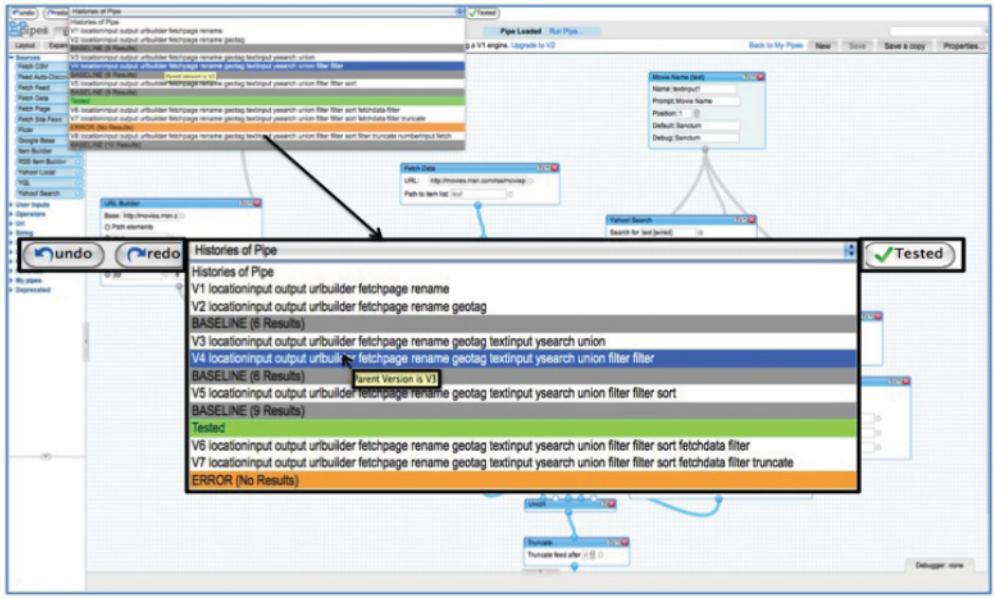

Fig. 3. Pipes Plumber interface.

consecutive versions of a pipe. Undo renders the previous version, while Redo renders the next version. The third widget, Tested, is a button that allows users to indicate that they have confidence in their pipe's correctness.

The fourth widget, History of Pipe, displays the modules added or removed for a pipe, per version, so that users can view the differences between versions. The History of Pipe widget, implemented as a drop-down list, allows a user to select a desired version from the list of available versions of the pipe. At present, the History of Pipe list presents versions in chronological order, and this may mask cases in which a version is actually derived from some much older version (e.g., V8 from V3). While we might be able to address this by employing graphical representations of the versioning history, initially we have addressed this by displaying further information on version provenance in tooltip boxes that appear when the pipe programmer hovers over a given version (Figure 3). To better describe the local history of pipes in terms of states and actions, we also provide textual descriptions of the contents of the versions.

In Yahoo! Pipes, debugging is largely performed by observing the runtime behavior of an executed pipe. To help users debug, we color code the version history in the History of Pipe list with execution results so that users can distinguish between successful pipe runs, unsuccessful pipe runs, and pipes that users consider to be "tested." Versions highlighted in grey (in Figure 3, the menu item labeled "BASELINE") are versions that were successfully executed (the pipe returns nonnull results). The numbers of results returned during that particular run are also shown. Orange (item labeled "ERROR (No Results)") signifies that a pipe execution returned no results, a probable indication of an error. Green (items labeled "Tested") indicates that the pipe programmer has confidence that a version is operating correctly, a status that can be applied by using the Tested button in our interface.

proxy wrapper issue commands appropriate to that versioning system. The implementation can also be broadened if services like Yahoo! Pipes were to incorporate a versioning tool into their system.

4. EMPIRICAL STUDIES

We have conducted two user studies. The first, an exploratory study, involved nine participants and focused on pipe creation and understanding tasks in the absence and presence of versioning support. The second study, a controlled experiment, involved 24 participants, and focused on pipe creation and debugging tasks in the absence and presence of versioning support.

4.1. Study I (Exploratory Study): Versioning for Creation and Understanding

Our first study addresses the following research questions:

—**RQ1:** To what extent does versioning allow mashup programmers to effectively and efficiently perform tasks related to mashup creation?
—**RQ2:** To what extent does versioning help mashup programmers understand complex third-party mashups?

4.1.1. Pipes Plumber Platform. For this study, we added basic versioning facilities in the form of "undo", "redo" and "History of Pipe" widgets to the Yahoo! Pipes interface as described in Section 3.

4.1.2. Participants. We decided to use expert programmers as subjects in this study because (1) if experienced programmers cannot make use of Pipes Plumber, then we expect it will be difficult for end users to make use of it; (2) these users are easily accessible; and (3) Yahoo! Pipes is popular with both professionals and end-user programmers. To recruit such participants, we sent an email to a departmental mailing list. As an incentive, participants were included in a raffle for a $25 prize. Nine students responded to our advertisement. All students were male, with seven from computer science or computer engineering and two from other departments. Four of the students were undergraduates and five were graduates. None had prior experience with Yahoo! Pipes, but all had some programming knowledge (78% knew multiple programming languages).

4.1.3. Independent and Dependent Variables. The independent variable in this study involved the presence (Figure 3) or absence (Figure 1) of versioning information. Participants interacting with our enhanced Yahoo! Pipes environment had access to the versioning help provided by that environment.[12] Participants interacting with the original Yahoo! Pipes environment did not have access to such versioning help because Yahoo! Pipes does not provide such support.

Dependent variables measured in this study were (1) the correctness of tasks and (2) the time required to create pipes. These variables allow us to measure the effectiveness and efficiency of the participants at performing their tasks.

4.1.4. Study Setup and Design. The study used a single factor, within-subjects design (a design in which the independent variable is tested with each participant [Wohlin et al. 2000]). We opted for a within-subjects design for three reasons. First, we wished to minimize the effects of individual differences among participants. Second, a within-subjects design allows us to gather more data using a smaller sample size. Finally, since each participant in our within-subjects study gained experience performing tasks using the environment with and without our versioning support, they were better positioned to provide feedback about the usefulness and usability of our versioning support than participants in a between-subjects study would be.

[12]In this initial study, "Tested" widgets and tags related to "ERROR" and "Tested" had not yet been added to the interface.

In this study, all participants attempted two pairs of two types of tasks, where the first task in each pair was attempted using the standard Pipes environment and the second was attempted using Pipes Plumber.

We used think-aloud protocol in our study, asking participants to vocalize their thought processes and feelings as they performed their tasks [Lewis 1982]. We used this protocol because a primary objective of the study was to explore participants' thought processes when using Yahoo! Pipes and our extension and gain insights into the barriers and problems that they faced. This protocol required us to administer the study to participants on an individual basis with an observer; in this case, the first author. The observer provided no input about the tasks to the participants.

We performed the study in the Usability Lab of the Computer Science and Engineering Department at the University of Nebraska-Lincoln. At the beginning of the study, participants were asked to complete a brief background questionnaire, which was followed by a tutorial of approximately 10 minutes on Yahoo! Pipes and versioning in general. The tutorial also included a short video of a sample think-aloud study so that participants could understand the process. After participants completed the tutorial, we asked them to create a small sample pipe to give them hands-on training. We also provided them with a list of documentation links that were within the Yahoo! Pipes web site.

Following these preliminaries, we asked participants to complete tasks for the study. Each participant completed a pair of tasks for RQ1 (mashup creation) followed by a pair of tasks for RQ2 (mashup understanding). The first of each pair of tasks was a task without versioning support (control tasks), and the second was a task with versioning support (experimental tasks). We also provided documentation on the versioning tool and requirements for the tasks to the participants; they were allowed to refer to these documents at any time while performing their tasks. We audio recorded each session and logged the users' on-screen interactions using a screen capture system (Morae[13]). The total time required for completion of the study per participant was approximately 1.5 hours, which included an average of 60 minutes for task completion.

After participants completed all tasks, we administered an exit survey. The survey consisted of both closed and open-ended questions about the tasks, the interface of our versioning extension, and the experiment process.

4.1.5. Tasks. We designed two tasks to address our research questions: Task 1 and Task 2. Task 1 was designed to allow us to observe reuse behavior and hence addressed RQ1, whereas Task 2 was designed to allow us to observe how participants comprehended a given pipe and was related to RQ2. Because the study was within-subjects, we further subdivided each task into two categories: Control (C) and Experimental (E) tasks. Therefore, in total we defined four tasks, with Task1.C and Task1.E addressing RQ1, and Task2.C and Task2.E addressing RQ2, respectively. We also provided participants with the instruction handout for Pipes Plumber.

Task 1 required a participant to create a pipe for a given set of requirements. Participants were first given a similar existing pipe (of 10 modules) as a template that they needed to understand. Then they were asked to create a pipe reusing some parts of the first pipe and adding some functionality. The existing pipe for Task1.C allowed a Spanish-speaking person to search for reviews of any business within a geographical location and within a certain distance, such that all results returned contained the search term in the title and were sorted in alphabetical order. Once a participant understood the pipe, they were asked to create a pipe that allowed a search for a review of any item around any area within a certain specified distance, but with the addition of

[13]Morae: http://www.techsmith.com/morae.asp.

new functionality; namely, the ability to change the original title of the search results to a title of their choice.

Task1.E involved a different pipe of complexity similar to the one used in Task1.C. This pipe was designed for a news enthusiast who wanted to search Yahoo! News for a topic filtered to display only those news items with some specified keyword in the title. The results of the query were to be unique and contain at least two items in reverse order based on the date of publication. As in Task1.C, participants were asked to understand the given pipe and then move on to the next step. In the second part of Task1.E, participants were asked to create a pipe that allowed a French-speaking user to search Yahoo! News for a search term while ensuring that search result titles were unique and translated into French.

Task 2 required participants to view a given pipe and answer a set of multiple-choice questions about the pipe and its functionality. The pipes used in Task 2 (Task2.C, Task2.E) were larger and more complex than those used in Task 1. Task2.C involved a pipe of 47 modules that displayed a list of unique, "mashed up" contents from five different sites specified by its user. A user could also limit the number of results that were displayed and sort results in descending order of date. Task2.E involved a pipe of 50 modules and was a generic filter used to merge feeds from different sources, and remove duplicate items received from those feeds. Further, a user could specify four different feeds, truncate or limit the maximum number of resulting items per feed, and select a maximum number of items that could be displayed.

For each of the foregoing tasks, participants in the Experimental group were provided with sample pipes that had versioning histories already associated with them. This was needed so that participants could investigate how each pipe was built from the ground up, and in this way, better understand the different functionalities provided by the modules in the pipe. To provide realistic versioning histories, the first author, given the intended specification for the pipes, created those pipes by following what, in their experience, represented reasonable appropriate programming increments, adding modules and employing tags that seemed appropriate for those versions. It is these versions that were initially available to participants in the History of Pipe information provided by Pipes Plumber. The versions for both tasks were designed to incrementally illustrate each feature or module working correctly, but in practice this situation may not hold; rather, a beginner may begin with no such information, working with a blank History of Pipe list.

The Control group received the same sample pipes as the Experimental groups, but without versioning information.

Measures

To evaluate whether versioning helped pipe programmers in their tasks, we measured the time it took participants to complete each task and the quality of the resulting pipes. We measured the duration of the time needed by participants to finish a task in minutes. We measured the quality of pipes by grading the pipes created by each participant to create a correctness score ranging from 0 to 100, where high scores indicate better performance. To reduce possible bias in the grading scheme, the first author and a graduate student not involved in the research worked together to create a grading scheme. They used this to grade pipes individually, and then they conferred and came to consensus on the grading results. The following paragraphs outline the grading scheme used.

On *Task 1*, participants could either reuse the modules from the pipe provided in the preliminary step or create the pipe by reimplementing the modules. In either case, they were already aware of the functionality provided in the given modules. We thus assigned 40 points to the correct use of the modules or the correctness of the functionality they provided. The remaining 60 points were awarded if the participant could

Table II. Mean (Median) Correctness Scores and Time-to-Completion

	Creation		Understanding	
	Ctrl	Exp	Ctrl	Exp
Correctness	94.2% (95.0%)	98.1% (100.0%)	65.0% (60.0%)	65.0% (60.0%)
Time (mins)	17.2 (14.3)	7.2 (6.7)	-	-

correctly create the additional modules needed to complete the task. Participants were penalized 10 points if they used incorrect logic (i.e., incorrect wiring between modules) or incorrect URLs. Participants were also penalized 5 points for each erroneous module added. Finally, for each module that remained in the pipe that had no meaningful impact on the output, two points were deducted. These penalties would have resulted in negative scores; to avoid this we set 40 points as the lower limit.

On *Task 2*, we measured only correctness. The task consisted of five quiz questions. Each question was worth 20 points; hence, a correct answer resulted in the assignment of 20 points and an incorrect answer resulted in the assignment of zero points.

4.1.6. Analysis Methodology. In this study, we asked our participants to perform experimental tasks after performing control tasks. A consequence of this choice is that in moving from the control task to the experimental task, participants might benefit from the former experience in a manner that might influence their results on the latter (learning effects). Furthermore, given that the study was conducted following think-aloud protocol, results involving time measures can be affected. Because of these design limitations, we rely primarily on our observations of how participants performed their tasks while using, or not using, versioning support. In our analysis, we refer to the participants individually as P[i] ($1 < i < 9$).

4.1.7. Results. Table II summarizes the overall results of our study. We next address our research questions in turn; we then discuss implications of the results.

RQ1: Versioning and Mashup Creation

To address our first research question we considered the correctness of pipes and the time required to create pipes with and without versioning support.

As our participants attempted Task1.C and Task1.E, we allowed them to proceed to the next task when they believed that they had finished creating pipes as per the given requirements. Because there was no acceptance test, some of the resulting pipes were partially incorrect. Participants were largely successful in creating correct pipes in both the control and experimental tasks, with mean correctness for the control (Ctrl) task being 94.2% (median 95%) and mean correctness for the experimental (Exp) task being 98.1% (median 100%). Participants exhibited a higher measure for correctness of pipes when engaged in the experimental task than when engaged in the control task.

As participants performed tasks relevant to RQ1 (Task1.C and Task1.E), we examined the total time required for them to complete their tasks, which included both the time required to understand and comprehend the given sample pipe and the time required to implement the required pipe. The mean time spent by all participants in Task1.C was 17.2 minutes (median 14.3 minutes), while the mean time spent for Task1.E was 7.2 minutes (median 6.7 minutes).

RQ2: Versioning and Mashup Understanding

Task 2 was designed to help us address our second research question regarding the benefits of versioning in enabling the understanding of complex, third-party mashups. In addition to observing participants' behavior in Task 2, we administered a multiple-choice quiz in which participants were asked to answer questions to help us assess the degree to which they understood the pipes. The mean correctness for the control (Ctrl)

task was 65% (median 60%), and the mean correctness for the experimental (Exp) task was also 65% (median 60%).

A post-hoc analysis of our participants' behavior showed that they often (in five cases out of nine) did not use the versioning features while engaged in the understanding task even when these features were available, so it is difficult to ascribe the lack of differences in understanding across tasks to the presence or absence of versioning.

4.1.8. Discussion and Implications. To obtain additional insights into the behavior that our study participants engaged in while performing their tasks, we analyzed the study data qualitatively.

Success in Reuse. One of our initial conjectures about the potential usefulness of versioning for mashups was that making earlier versions available would enable mashup programmers to better reuse mashups. Since reuse is one of the primary mechanisms by which end-user programmers construct new programs [Cypher et al. 2010], data on this premise might further inform the mechanisms by which we provide versioning. Our exploratory study indicates that versioning can facilitate the reuse of pipes and modules.

The reuse task can be further divided into two subparts: (1) understanding the functionality of the given pipe and (2) reusing parts of that pipe to create a pipe with the required functionality. We studied data relevant to each of these subparts to investigate whether the presence of versioning had an effect. We discuss the understanding subpart later in this section (along with discussion of Task 2).

We found that a key reason for the success of the Experimental group (Table II) was the availability of the evolution history of the pipe through the History of Pipe list. This list provided a step-by-step guide about which modules were added to the pipe and the resulting output, which in turn helped Experimental group participants select the parts of the pipe that they needed to reuse. For example, while performing Task1.E, participant P7 commented, "[Looking at History of Pipe] version 2 seems to be the one needed [pauses while still looking at History of Pipe] version 3 has the filtering which is not needed ... so ... I will take version 2 and add to it. ..." Participant P7 commented: "what versions are there ... [looking at History of Pipe] ... we need the `textinput`, `output`, `yahoo search` and ... looks like we don't need `sort` and hence we need the version 2."

This benefit of versioning was suggested in the participants' feedback in our exit surveys. Eight of nine participants mentioned that they thought versioning improved the reusability of pipes and all eight had used versioning in their experimental tasks.

Success in reuse also translated into efficiency gains. In the experimental task, because participants were able to identify the most appropriate version upon which new functionality was to be added, they saved time that would otherwise have been spent examining and removing extra modules. For example, participant P7 spent 4.41 minutes performing the reuse task in the presence of versioning, compared to 31.17 minutes without versioning.

Identifying and Locating Pipes. Currently, there are no mechanisms for identifying the ancestor of a pipe (the *initial pipe* from which a subsequent pipe is derived) in the Yahoo! Pipes environment. Moreover, one can recognize that a given pipe has been cloned, but there is no way of identifying which modules belonged to the original (*parent*) pipe from which the cloned pipe was obtained, and which modules are new. Because of these constraints, many participants in the Control group faced problems. We found that participants had trouble locating initial or parent pipes for reference during tasks and identifying the modules that they had added versus those that were from the parent pipe. For example, participant P6 in Task1.C cloned the given pipe and started editing, but when he became stuck and wanted to restart his work, he was

unable to locate the original pipe from the list of pipes that Yahoo! provides. Similarly, participants who cloned a pipe often removed too many modules from it or did not remember how the modules that they had removed functioned in the parent pipe. As an example, participant P7 was unsure about how to replace all of the item titles in a pipe (Task1.C). He was unsure about whether to use and how to use the `loop` module and experimented with the module by adding and deleting it multiple times. To help with such problems, some participants (P1, P4, and P7) viewed the initial pipe by opening it in a separate window. In all cases, however, if participants had versioning available, the original (and subsequent work-in-progress) pipes would have been saved and been readily available—a fact that all participants noted in the exit survey.

Backtracking to Prior Versions. We found that during their programming tasks, in addition to seeking information on initial or parent pipes, participants often desired to return to earlier "intermediate" versions of pipes. In other cases, participants deleted more modules than required and wished to bring those deleted modules back. In both situations a versioning tool could have been helpful. For example, while performing Task1.C, participant P8 deleted most of the modules and then wished to return to an earlier version, commenting: "I'm just trying to get back [to an earlier version] so I can see how this pipe works."

While providing access to prior versions appears to be important, the manner in which the access is provided also matters. We observed that most participants started editing the given pipe from the second version (V2). However, the next version saved by the participants was version seven (V7), as our system generates and presents versions in a linear order, and the pipe provided to participants already had six (prior) versions. This caused the parentage information on the newly cloned pipes to be hidden, causing some confusion for participants; a factor that led us to add tool tips with parentage information in subsequent enhancements to the interface.

Debugging: We observed that the messages generated by Yahoo! Pipes as participants debugged and tested their changes were difficult for them to comprehend. As a result, they were often forced to revisit their earlier versions to find errors. For example, participants often made mistakes while performing the task that required them to add two additional modules to a pipe. Participant P8 was not happy with the results of his actions and removed the `loop` and `string replace` modules, replacing them with the `rename` module. He made a couple more changes and checked the debugger output to observe the runtime behavior of the resulting pipe. After a few minutes, he realized that his modifications were not correct and commented, "Umm, actually it was working with the string replace, tyring to figure out how to get that to work." Later he commented, "So, I think I broke something actually. I gotta figure out what." Providing mechanisms by which users can tag specific versions, to facilitate later return to them, could help with such problems.

Pipe Understanding. In Task 1, versioning helped participants understand the structure and rationale underlying the third-party pipes and their evolution. The History of Pipe list provided a structured mechanism that allowed participants to understand the pipe functionality incrementally. This was evident when participant P4 commented, "You can see how each pipe was developed along the way to final product." Another participant, P5, stated in the exit survey that versioning was helpful when trying to learn about the functionality of a pipe: "When looking at the overall finished pipe it was quite intimidating. Using the versioning tool, it was much easier to follow along the order of the finished pipe to gain a better understanding."

In Task 2, in contrast, we found that the impact of versioning on pipe understanding was somewhat limited (Table II). This might be due to the following reasons. First, this task contained 45 modules and had a long history of evolution. A prior study has shown that typical pipes in the repository have sizes ranging between 6 and 8 modules

[Stolee et al. 2011]. Therefore, it is possible that participants might have been overwhelmed with the amount of complexity in this task. Second, it may be possible that our Task 2 quiz results are more reflective of the difficulty in measuring the participants' ability to understand a particular pipe than of their use of versioning. Finally, unfamiliarity of the participants with the functionalities of our interface might have also slowed them down.

4.1.9. Enhancements to Tool and Study Design. The results of our exploratory study prompted us to investigate several enhancements to our versioning support, and to design a more controlled experiment to study this support.

Tool Design. We found that debugging is difficult and was a key issue in our first study; we therefore wished to investigate whether our versioning support can help users perform debugging tasks. While conducting Study I (Task 1), we observed that participants often used the History of Pipe widget, and that they often wished to refer back to specific prior (working) versions. This motivated us to provide markers to distinguish erroneous versions from tested versions of pipes, and to indicate when users believe that pipes have been adequately "Tested"—as discussed in Section 3. We also wished to assist users in finding prior versions in cases where parentage was masked; this led us to add parentage information through tool tips.

Study Design. There were three primary changes we wished to make to our study design. First, we wished to investigate a more diverse population of users, including both Computer Science Experts (CSEs) and End Users (EUs). End users are different from computer science programmers because they program for personal use rather than public use [Ko et al. 2011]. Examples of end users are wide ranging; they include scientists, engineers, interaction designers, web masters, actuaries, and so forth. End users also differ from professional developers in terms of the amount of experience they have with software engineering concepts.[14] Professional developers, in contrast, are more concerned and hence pay more attention to quality issues of their software.

Second, we wished to investigate the use of versioning support in a manner that lessens the limitations of our exploratory study with respect to possible learning effects, and in a manner that enables more precise estimations of time by eliminating the think-aloud process from the experiment and counterbalancing the tasks.

Third, because we discovered that participants found debugging difficult in Yahoo! Pipes and we wanted to test our enhancement to the History of Pipe widget, we replaced the "understanding" task with a "debugging" task. (Note that the reuse task already included an "understanding" component.)

4.2. Study II (Controlled Experiment): Versioning for Creation and Debugging

Our exploratory study provided support for the claim that versioning can help mashup programmers create mashups more efficiently and effectively. In our second study, we wished to further investigate this claim with a more diverse population and in a manner that enabled more precise estimations of time by eliminating the think-aloud component. We also wished to investigate whether our versioning support can aid users in performing debugging tasks. We thus framed the following research questions:

—**RQ1:** To what extent does versioning help mashup programmers reuse and debug mashups effectively and efficiently?
—**RQ2:** To what extent does versioning render reusing and debugging tasks different for mashup programmers in terms of effectiveness and efficiency?

[14]Software engineering is defined as "the application of systematic, disciplined, activities that address software quality issues" [Ko et al. 2011].

Table III. General Demographics

Characteristics	Distribution		
		CSE	EU
Gender	Male	12	10
	Female	0	2
Age	19–23	8	7
	24–29	4	3
	30–40	0	2
Education	Undergrad	7	7
	Grad	5	5
Number of Known Languages	1–2	1	10
	3–5	10	1
	6–10	1	1
Programming Experience	None	0	2
	<1 year	3	5
	2 years	1	2
	3 years	3	2
	4 years	1	1
	>4 years	4	0
Yahoo! Pipes	Familiarity	1	1
	No Familiarity	11	11

—**RQ3:** To what extent and with what differences does versioning benefit computer science participants and end users?

We conducted a pilot study of three participants from our lab to help us adjust the experimental artifacts and processes, results of which are not discussed here.

4.2.1. Participants. We recruited 24 students from the University of Nebraska–Lincoln by sending emails to student mailing lists across departments at the university. Participants were selected from those responding to our email on a first-come, first-served basis, and were paid $20 for their participation in the study. Table III summarizes the general demographics of our participants. Twelve of the participants were from the Computer Science and Engineering Department and had the formal training in programming expected of students in such a program; we refer to these students as "CSE participants." The other 12 students were from other departments and had no formal training (at least, not to the level of computer science students) in programming. Several of these end users ("EU participants"), however, had programming training appropriate to the more rudimentary requirements of their majors, such as in the use of introductory Matlab, C, or C++ for non-CS majors. The distribution of the majors among the the latter students was Electrical Engineering (2), Mechanical Engineering (1), Civil Engineering (4), Transport Engineering (1), Biological Systems (2), Materials Science (1), and Actuarial Science (1). Participants' ages ranged from 19 to 40 years. Of the participants, 22 were male and 2 were female; these were distributed across the two treatment groups. Only two of the participants had prior experience with the Yahoo! Pipes environment, and neither of them had created more than five pipes.

4.2.2. Independent and Dependent Variables. Our independent variable involved the presence (Figure 3) or absence (Figure 1) of versioning information, with the former provided via Pipes Plumber. To measure effectiveness and efficiency, we used two dependent variables tracking (1) the correctness of pipes following creation or debugging activities, and (2) the time required to create or debug a pipe.

4.2.3. Study Setup and Design. In most respects, our study setup and design followed that used in Study I (the primary exception being counterbalancing of tasks, discussed in the following text). We used a mixed-methods design, with task type (creation, debugging) and environment (Pipes, Pipes Plumber) as within-subjects variables and programming expertise (CSE, EU) as a between-subjects variable.

We conducted the study in the Usability Lab at the University of Nebraska–Lincoln, where we could observe and record transcripts for each participant. We audio recorded the sessions and logged the participants' on-screen interactions using Morae. The study protocol was administered to each participant individually, with the first author conducting each session but providing no input to participants while they were engaged in their tasks. The total time required for completion of the study was approximately 2 hours, which included about 60 minutes for task performance. After all tasks were completed, we administered an exit survey to obtain further feedback.

4.2.4. Tasks. Study 2 included two types of tasks. Task 1 required participants to create pipes, given pipes that they could choose to reuse portions of. Task 2 required participants to debug pipes. To enable a within-subjects study we again created two distinct subtasks for each type of task (reusability and debugging) so that we could examine the participants' experience with and without versioning for that type of task.

Task 1 was followed by Task 2 for all participants. We did not counterbalance Task 1 and Task 2 because Task 2 (debugging a faulty pipe) was a more complex task and we believed that it was beneficial for participants to gain some expertise with the Yahoo! Pipes environment to reduce understanding barrier effects (see Section 5), which would enable participants to be more successful in Task 2 and avoid frustration. Note that this learning process does not bias our results, because all participants had similar experiences with and without versioning support in Task 1.

Subtasks within Task 1 and Task 2 were counterbalanced in the following manner: Within each task, we performed two levels of counterbalancing. First, we counterbalanced the order in which participants received the treatment; that is, half of the participants performed the control task before the experimental task and the other half performed the experimental task before the control task. Furthermore, to ensure that there was no bias associated with a treatment, we counterbalanced the subtasks; that is, half the time subtask "search" was associated with the control treatment and subtask "blog" with the experimental treatment and the other times subtask "search" was associated with the experimental treatment and subtask "blog" with the control treatment. We performed similar counterbalancing for subtasks ("eBay" and "movie") for Task 2.

As in Study 1, each of the pipes provided to participants in the Experimental group was provided with versioning histories. Again, the first author created the versions for these pipes based on what, in her experience, represented appropriate programming increments, employing tags that also seemed appropriate for those versions. (Note that in practice, a beginner creating a pipe from scratch would begin with a blank History of Pipe list, and have no such information.)

During the study, we provided the participants with various documents. These included specifications and requirement documents for each subtask, and the output of the correct pipe to act as an oracle. Participants were also given an instructional handout for Pipes Plumber.

Next, we describe each subtask in detail.

Task 1

Task 1 involved two steps. The first step required a participant to understand the functionality of a given pipe. The second step required them to create a pipe that had some functionality in common with the given pipe. They could complete this task either

by creating the new pipe from scratch or by reusing some of the modules in the given pipe. The two given pipes were of similar complexity and involved similar numbers of modules (13–14 each). The two subtasks were as follows.

In *Task1.search,* participants were given a pipe that lets users search for a review of any item within a given distance of a given location (e.g., reviews of museums within 10 miles of Lincoln, Nebraska). The distance from the location is at the discretion of the user and hard-coded in the pipe. The user can choose the number of reviews to be displayed. The results of the search are translated into Korean. The titles of the reviews are unique (i.e., duplicates are filtered out) and they are presented in alphabetical order.

The next step required participants to create a pipe similar to the one used in the first step. The difference was that this step required participants to implement extra functionality; namely, to rename all of the titles in the search result with a title of their choice (e.g., if a participant specifies the title to be "museums" then he or she will see all titles replaced by "museums"). Further, other actions such as sorting results, filtering out duplicates, and translating into Korean were not required of this pipe.

In *Task1.blog*, participants were given a pipe that allows a user to search for any topic in a set of blogs hosted on websites such as Blogpulse and Technorati. The pipe then merges the results into a single feed, displaying only unique results and sorting them in ascending order. It also allows the user to truncate the number of results based on user input at runtime and allows users to filter out items in which they are not interested (e.g., filter out crime items from the search results). Finally, the pipe provides a status display regarding the total number of results generated by the query and the truncated results.

In the next step, participants were asked to create a pipe similar to the one used in the first step, but with the additional requirement that the results be translated into Spanish. As in the previous task, the results of this pipe were not required to be unique, sorted, or filtered on a given keyword. The display status from the previous step also was not required and was considered extraneous.

Task 2

In Task 2, participants were given pipes into which we had seeded two faults. These faults were related to *missing module, missing parameter value* and *incorrect module* problems in the pipe. Each pipe included 15–16 modules. Participants were given detailed requirements about the functionality of the correct pipe along with an example of the output of the correct pipe. Participants were required to correct the seeded faults and ensure that the pipe was working as stated in the requirements by matching their results with the sample output. The two subtasks were as follows.

Task2.movie involved a pipe that allows a user to generate a list of local theaters by inputting their zip code. The pipe then collects a list of movies and displays them, together with show times and geolocations, on Yahoo! Maps. The user could also obtain a poster and reviews of a movie. This pipe included *missing module* and *missing parameter value* faults.

Task2.eBay involved a pipe that allows a user to search for an item on an auction site such as eBay, or in a list of classified ads on Craigslist (e.g., San Francisco Craigslist). The pipe allows the user to set a price range by inputting minimum and maximum amounts. Search results include the name of the site from which the item was retrieved. The user has the option to limit (truncate) the number of results displayed. This pipe included *missing module* and *incorrect module* faults.

Measures

To evaluate whether versioning helped pipe programmers in their tasks, we measured the time participants spent performing tasks and the correctness of the resulting pipes. We measured time in minutes and measured correctness using scores ranging

from 0 to 100. On *Task 1*, we used the same measures used to judge the correctness of pipes in Task 1 in Study I. On *Task 2*, 80 points were given if participants successfully identified and corrected the seeded errors in the given pipe. There was no partial credit for just identifying the error. The remaining 20 points were allocated to other errors; specifically, if participants added new errors when implementing a pipe, they were penalized 10 points, and if they added unrelated modules, they were penalized five points apiece (up to a maximum of 10 points). This penalty could have resulted in negative scores; to avoid this, we set 20 points as a lower limit. If participants created correct pipes, they were given 100 points.

4.2.5. Threats to Validity. Where external validity is concerned, our participants were all university students, and the 12 who constituted our "end-user" population were primarily engineering students. Further, almost all participants were male. A second threat to external validity is that our study considered only two tasks that built on only two types of pipes. Additional studies of different populations, tasks, and pipes are needed. A third threat arises from the fact that the participants were asked to use pipes that were provided, rather than pipes which they had created for themselves. While the reuse context is common and important, prior familiarity with pipes and histories could lead to different results. A fourth threat arises from the fact that the histories provided with the pipes given to Experimental groups, and the bugs placed in pipes, represent only a small sample of the possible histories and bugs that could be associated with pipes. These threats can be addressed only through further studies.

The primary threat to internal validity for this study relates to our choice of a within-subjects design. This study design helped minimize effects related to individual differences and the use of a small pool of participants, but it might have led to learning effects as participants moved from initial to later tasks. In this study, however, unlike the first, we counterbalanced the order in which participants were assigned to the Control and Experimental groups, by randomly selecting participants to either perform the experimental task and then the control task or vice versa.

A second internal validity threat can arise if our pairs of subtasks (two per task type) are not of equal complexity, in which case results might be dependent on subtask. To limit this threat, we interchanged the subtasks that were given to the Control and Experimental groups (i.e., Task1.search was used as a control task on half of the runs and as a treatment task on the other half). To assess whether any problems related to this threat had occurred, we performed paired t-tests for Task1.search versus Task1.blog and Task2.movie versus Task2.eBay, ignoring order and tool type, and found no significant differences. A third validity threat can arise from learning effects that occur between participants' performances of Task 1 and Task 2; however, we believed that the debugging task needed to be performed second in all cases as debugging is more difficult and participants would benefit from learning about the environment during Task 1.

Construct validity threats include the possibility that the complexity of our pipes was not high enough to allow measurements of effects and that the pipes used for control and experiment tasks were not comparable in complexity. We controlled for this by performing a pilot study of three end users and using their feedback to adjust the pipes and the tasks. Further, as already reported, we found no significant differences when we compared performance across subtasks.

Where conclusion validity is concerned, correctness of pipes was measured manually and this could be a source of bias; however, as noted, we took steps to limit the likelihood of such problems by employing a graduate student outside the research group.

Table IV. Correctness Results

Effects	DF	F value	p-value
Tool type	44	24.04	<**0.0001**
Task type	22	5.35	**0.0304**
Expertise	22	2.19	0.1530
Expertise by Tool type	44	1.06	0.3094
Expertise by Task type	22	0.12	0.7326
Tool type by Task type	44	0.07	0.7937
Expertise by Task type by Tool type	44	3.75	0.5920

Table V. Time Results

Effects	DF	F value	p-value
Tool type	42	6.50	**0.0145**
Task type	20	1.66	0.2120
Expertise	20	0.61	0.4453
Expertise by Tool type	42	1.52	0.2241
Expertise by Task type	20	3.72	0.0681
Tool type by Task type	42	0.10	0.7592
Expertise by Task type by Tool type	42	0.03	0.8607

4.2.6. Analysis Methodology. Because our data depended on three main factors (Expertise, Tool type, and Task type), we conducted a three-way analysis of variance on our data, treating one factor as between subjects (Expertise) and two factors as within subjects (Tool and Task). Therefore, our analysis had three main effects (Expertise, Tool type, and Task type) and four interaction terms (Expertise by Tool type; Expertise by Task type; Tool type by Task type; and Expertise by Tool type by Task type). We used split-split plot ANOVAs [Dowdy et al. 2004] in order to consider all the error terms in the statistics. We performed these ANOVAs using the SAS programming language [Cody 1988].

We analyzed our data for correctness and time separately.[15] We also qualitatively analyzed the participants' transcripts to gain additional insights into their behavior. In our discussions, we refer to end user participants as EU[i] and computer science participants as CSE[i] ($1 < i < 12$), respectively.

4.2.7. Results. We found that the three-way and two-way interaction effects for correctness as well as time were not significant at $\alpha = 0.05$ (see Tables IV and V). Therefore, we investigate only the effects of the main factors when answering the research questions.

RQ1: Effects of Versioning

To address our first research question we considered the correctness of the results when participants performed reuse and debugging tasks with and without versioning support, and the time taken to complete the tasks in those contexts.

As in Study I, we allowed our participants to proceed to the next task whenever they wished to. There was no acceptance test or time limit attached to the performance of any of the tasks. The results of our analysis (at $\alpha = 0.05$) show that the main effects of the tool for the correctness ($p < 0.0001$) and time ($p = 0.0145$) measures were statistically significant. This suggests that the availability of our versioning support helped our

[15]During the experiment, two EU participants in the Experimental group could not find the pipe on which they were working in the list of pipes (provided by Yahoo!); because of this, they lost their changes and needed to redo their tasks from scratch. Because we create versions only when users save their pipes, versioning did not help these participants recover their lost code; hence, we exclude them from our sample where time costs are concerned.

Table VI. Participants (Experimental Group) Who Checked Versions during Task Completion

Activities	Instances of versions checked	
	Task 1	Task 2
undo	4	5
redo	1	3
inspect list view	61	145

Table VII. Mean (Median) Correctness and Time Scores with and without Versioning

	Reuse		Debug	
	Ctrl	Exp	Ctrl	Exp
Correctness	65.42% (70%)	87.92% (90%)	53.33% (60%)	73.54% (85%)
Time (mins)	18.14 (13.58)	14.30 (11.14)	20.53 (18.18)	15.73 (12.02)

participants reuse and debug pipes. The Experimental group was significantly better in providing more correct results and took less time than when participants did not have access to that support.

All participants in the Experimental group used versioning support. In fact, in cases in which participants had versioning support as the first treatment, they requested that support when they were required to work without it (when working in the Control group). Further analysis of participants' versioning use (see Table VI) shows that few of them used the undo/redo functionality; instead, they predominantly used the History of Pipe list. This list allowed them to view the provenance of their pipes along with their "testedness."

Versioning and Mashup Reuse

The mean correctness scores for the mashup reuse task were 65.42% (median 70%) without using versioning and 87.92% (median 90%) when using versioning. The mean time costs for the mashup reuse task were 18.14 minutes (median 13.58 minutes) without using versioning and 14.30 minutes (median 11.14 minutes) when using versioning (see Table VII). Our results show that versioning helped participants reuse parts of pipes.

Participants in both treatment groups began their tasks by first studying the specifications (written descriptions provided to them) of pipe behavior. Then, they attempted to comprehend the pipe code by inspecting modules. To obtain a sufficient understanding of their pipes, participants also needed to understand the text or source contents of the websites that were used as data sources in the pipes. Such understanding would have helped them comprehend the structure as well as contents of the websites, and helped them determine which components of those sites are used as parameters in pipe modules. However, while attempting to comprehend pipes, only one participant (EU12) examined relevant websites. Other participants investigated the given pipe by focusing only on the modules and their functionality. Once participants believed that they had sufficient understanding of pipe functionality, they moved on to their assigned task of reusing a pipe.

After studying the requirements of the tasks, participants decided either to work on the same pipe, create a new pipe or "clone and own" (copy) a pipe. In the Control group, most (10 of 12) participants cloned pipes while a few (2 of 12) created pipes from scratch. From our observations, it appears that these two participants wanted a "fresh start." They opened a new browser window and began working on their new pipe, while keeping the given pipe open in another browser instance. In contrast, in the Experimental group, some (25%) participants cloned pipes while most (75%) preferred

to work on the original pipes. We believe that a majority of participants started editing the given pipe because they knew that they could revert back to it at any time.

We attribute the success of the Experimental group to the fact that most of the participants (21 of 24) were able to use the History of Pipe list to select prior versions that contained the required functionalities. For example, while working in the Experimental group, EU3 required just 10 minutes to create his pipe correctly, but while working in the Control group, he spent 22.5 minutes without success. This difference occurred because the selected version acted as a template, allowing him to make just minor changes to make the new pipe work.

We also noted that participants "debugged their pipes into existence" [Rosson and Carroll 1993]; that is, their designing was interlaced with coding, testing, and correction activities. Typically, participants created their pipes by bringing in modules and seeing whether the pipe behaved as desired; if not, they then removed certain components. Another behavior we observed was opportunistic programming [Brandt et al. 2009]; behavior that could help users explore different ideas quickly. Participants formed various and different ideas about how to implement solutions—ideas that had different strengths and weaknesses. Participants implemented solutions for those ideas, backtracked through their prior changes, and implemented alternative solutions.

In summary, participants in the Experimental group benefited from versioning when performing opportunistic programming or debugging programs into existence because they could revert back to prior successful or unsuccessful versions and attempt to pursue different strategies. In the exit survey, participant EU6 commented: "Nice to be able to go back to the old files easily." Participants in the Experimental group were also more adventurous and explored more ideas than participants in the Control group, and we attribute this to the fact that they knew that they could return to prior versions. Participant EU8 confidently tried four different ideas, noting that he was aware that versioning would allow him to return to prior versions. In the control task (which occurred after his experimental task), he asked the observer for versioning support, inquiring: "Where is the version history?"

As noted in reference to Study I, some of the participants also referred to prior versions as examples and used them to understand the usage of components or connections. For example, while performing Task 1, participant CSE1 wished to use the `string builder` module, so he studied prior versions to see how to use the module and then was able to use it correctly. The History of Pipe list allowed participants to identify the (pipe) version in which the module was used. The list also helped participants in the Experimental group understand the process by which pipes were created. In the exit survey, participant EU3 commented that with this list, it was "easier to see the process and flow of development."

Versioning and Mashup Debugging

The mean correctness scores for the mashup debugging task were 53.33% (median 60%) without using versioning and 73.54% (median 85%) when using versioning. The mean time costs for the mashup debugging task were 20.53 minutes (median 18.18 minutes) without using versioning and 15.73 minutes (median 12.02 minutes) when using versioning. Details are provided in Table VII.

In the Control group, participants executed their pipes and tested them for errors. As in Task 1, in Task 2, they also first attempted to understand the requirements for the pipe. In Task 2, however, the act of understanding the pipe overlapped with the activity of locating bugs. Since existing mashup environments allow only runtime observations [Grammel and Storey 2008], as participants explored the modules in the pipe, they also looked for errors by checking the debugger window for error messages. This strategy, however, required substantial time. For example, CSE6 inspected each module of his pipe and spent 40.13 minutes debugging the pipe without success.

Table VIII. Mean (Median) Correctness and Time Scores for Reuse and Debugging Tasks

	Reuse	Debug
Correctness	76.67% (79.00%)	63.44% (60%)
Time (mins)	16.73 (13.26)	18.06 (16.30)

In the Experimental group, the History of Pipe list facilitated bug localization. The tags for each run in the list indicate whether a specific version was untested, successful or unsuccessful, and allowed participants to judge the "testedness" of versions. This helped them narrow down their evaluation space. For example, in the Experimental group, participant CSE6 needed only 9.13 minutes to correctly debug the pipe. He looked at the version that was tested (V5) and those that were marked as erroneous (V6 and V7). By observing these, he was able to locate the two faulty modules in the pipe.

After participants located modules containing errors, they proceeded further, attempting to fix those errors. In the Experimental group, although participants were able to locate bugs from the History of Pipe list, they spent substantial time searching for fixes for those bugs. Most of the participants struggled to correct pipes. Note that no points were given if a fault was correctly located but not fixed. While fixing bugs, eight participants looked at documentation for example pipes, or versions (in the case of experimental tasks), containing similar modules to attempt to understand the usage of the pipes. Two participants searched (on Internet) the error messages or names of modules in an attempt to understand problems further.

As in Task 1, all participants pursued various ideas involving the use of different modules and parameters to attempt to correct errors. In the Experimental group, version management support allowed participants to try different ideas more effectively, as they were able to return to older (successful or unsuccessful) versions. For example, in the Experimental group, participant EU1 located an error in the `Fetch data` module and attempted to fix it using various alternative modules (`Fetch auto discovery` and `Fetch feed`). He also pursued different ideas, that is, bringing two `Fetch feed` modules in and connecting them with the pipe. In the exit survey, participant CSE1 commented on the helpfulness of this, saying: 'Developers can always go back to the previous version to fix a bug."

RQ2: Effects of Task Types

To address our second research question, we considered the time required to perform tasks of reusing and debugging pipes, and the correctness of the results.

The results of our analysis (at $\alpha = 0.05$) show that the main effects of tool for correctness measures were statistically significant ($p = 0.0304$), while the main effects for time measures were not significant ($p = 0.2120$). Hence, the correctness results achieved in reuse and debugging tasks differed significantly, with participants faring better in reuse tasks than debugging tasks, even though the times spent by participants in those tasks did not differ significantly.

The mean correctness score achieved in the mashup reuse task was 76.67% (median 79.00%) and the mean correctness score achieved in the debugging task was 63.44% (median 60%). The mean time cost for the mashup reuse task was 16.73 minutes (median 13.26 minutes) and for the debugging task was 18.06 minutes (median 16.30 minutes). Details are given in Table VIII.

The foregoing results might be seen as unsurprising, given that computer science as well as end-user programmers reuse their own or other's code quite often [Cypher et al. 2010]. In contrast, debugging is known to be difficult for professionals [Rosson and Carroll 1996] as well as end users [Cao et al. 2010b]. In our exit survey, participant CSE1 commented that "it's hard to debug," and similar views were echoed by participant EU12: "Last two tasks (debugging) were harder."

Table IX. Mean (Median) Correctness and Time Taken Scores for Reuse Task

	Correctness		Time (mins)	
	Ctrl	Exp	Ctrl	Exp
CSE	78.3% (78.0%)	87.9% (87.5%)	20.2 (12.8)	13.0 (11.1)
EU	52.5% (67.5%)	87.9% (93.0%)	16.1 (15.5)	11.4 (12.9)

We found that the debugging support provided by Yahoo! Pipes-observation of runtime behavior-was inadequate in helping participants debug correctly. For example, despite the fact that in the Experimental group participants had a smaller evaluation space (since participants were aware of the versions that were working correctly) they struggled to correctly fix bugs. While versioning support helped participants focus their debugging efforts and led to quicker fault localization, this step alone was not enough since the main challenge was in fixing the errors.

To fix bugs, participants had to investigate three possibilities: (1) whether the correct information source was being used, (2) whether the correct module was used, and (3) whether the module used was the appropriate one for the information source used. For example, the module `Fetch Feed` is used to display web content and requires the use of RSS feeds, whereas the module `Fetch Data` performs a similar step but requires XML content. One of the seeded faults involved the incorrect use of a module, that is, a `Fetch Data` module was used to read RSS feeds. The correct solution required the use of a `Fetch Feed` module. Identifying this mismatch and then finding the right information source for the module proved to be extremely challenging to participants.

We posit that participants found it difficult fix errors for several reasons. First, the error messages generated by Yahoo! Pipes were often difficult to interpret because they involved technical jargon (see Section 5.1). Second, the available help was often inadequate in pointing to the correct information source and the required format. Finally, the fact that mashup environments rely on external data sources and provide a black-box module-based visual editor added a layer of abstraction that made it difficult for participants to understand how the information sources were consumed by the modules (see Section 5.1)

As the foregoing discussion shows, the difficulties faced by participants when debugging motivates efforts to create additional debugging support for mashup programming environments.

RQ3: Comparing End-User and Computer Science Participants

To address our third research question, we considered the time required to perform reuse and debugging tasks with and without versioning, and the correctness of results in those contexts, as it relates to differences observed between computer science and end-user participants.

The results of our analysis (at $\alpha = 0.05$) show that the main effects of expertise for correctness ($p = 0.1530$) and time ($p = 0.4453$) measures were not significant. This indicates that computer science and end users did not significantly differ in terms of the correctness of the pipes they created during the reuse and debugging tasks. There was also no difference in the time costs. These results indicate that versioning was beneficial to both computer science and end users equally. Therefore, we focus only on qualitative differences observed in the behaviors of the two classes of participants.

Comparing CSE and EU Behavior While Reusing with Versioning

The details about mean correctness scores and time-to-completion for CSE participants and EU participants are provided in Table IX. Despite the lack of computer science background, EU participants produced pipes of quality comparable to those created by CSE participants when using versioning.

We found that EU participants were more open to seeking help from versioning support (viewing prior versions), whereas CSE participants (perhaps due to higher

Table X. Mean (Median) Correctness and Time Taken Scores for Debugging Task

	Correctness		Time (mins)	
	Ctrl	Exp	Ctrl	Exp
CSE	55.8% (60.0%)	80.0% (100.0%)	18.6 (17.6)	11.9 (10.4)
EU	50.8% (60.0%)	67.1% (67.5%)	22.4 (18.1)	20.5 (17.6)

self-efficacy) attempted to resolve problems on their own through trial and error. Only 9 of 12 CSE participants used versioning support. CSE participants preferred to create pipes from scratch and hence spent more time and effort in creating them. Therefore, it seems that versioning support can help end users learn by providing examples of correct structure and use.

We also observed that EU participants repetitively referred to example pipes to perform their tasks. Providing a means for users to refer to and execute prior versions of pipes while not losing their changes can, therefore, benefit end users. CSE participants, in contrast, were better at understanding example pipes and referred back to the given pipes fewer times than their EU counterparts.

Comparing CSE and EU Behavior While Debugging with Versioning

Table X provides the mean correctness scores and times-to-completion for the debugging tasks between CSE and EU participants.

We observed that all 12 CSE participants used the History of Pipe list to visually identify the modules that could be erroneous, whereas EU participants inspected and executed individual versions from that list. Hence, in the Experimental group, CSE participants were able to isolate problem modules and debug them to arrive at correct pipes with far fewer instances of executing pipe versions. We also found that EU participants needed to spend more time debugging that CSE participants to arrive at the correct output even after isolating the problem modules. Another difference is that CSE participants primarily used runtime observations to arrive at correct outputs, whereas EU participants also relied heavily on help mechanisms.

5. A QUALITATIVE ANALYSIS OF THE USE OF VERSIONING

One of the key goals of our study was to evaluate the usefulness of versioning support in the domain of mashup creation. Since ours is one of the first attempts we are aware of to study versioning support in an end-user domain, we wanted to qualitatively investigate the role and impact of versioning in enabling users to reuse and debug programs in a web mashup domain, namely Yahoo! Pipes. Our qualitative analysis provides insights into the challenges that users face when reusing and debugging pipes and the usefulness of versioning in overcoming these challenges. Our overall goal was to investigate the following research question:

—**RQ:** How can versioning benefit programmers in (1) reducing learning barriers, (2) helping them reuse parts of pipes, and (3) improving their debugging performance?

We analyzed the results of our experiments by first transcribing the experiment videos and interviews. We then coded the transcripts using three primary code sets: (1) programming barriers that participants faced while performing their tasks (Table XI), (2) reuse activities (Table XIII), and (3) debugging strategies (Table XV). We describe these code sets in the following sections. For each code set, two of the authors coded small portions of the transcripts independently and compared intercoder agreement until they reached an 80% agreement covering at least 20% of the transcripts. Once such an agreement was achieved, the first author coded all the remaining

Table XI. Programming Barriers Adapted from Ko et al. [2004a] and Cao et al. [2010b]

Barriers	Definition	Example
Understanding	Not knowing why the program behaved the way it did	User is unable to understand the type of error and its cause when viewing the output (of a module) in the debugger window
Use	Not knowing how to use a module	User did not know how to set parameters in the `loop` module
Coordination	Not knowing how to connect modules	User tries to connect text input module with count module, which are incompatible because of incompatible output and input types
Selection	Not knowing which module to use for a particular behavior	User incorrectly selects `rename` instead of `string replace` module while appending a prefix to titles of search result
Design	Not knowing how to frame the problem and arrive at a solution	User performs more work than needed, by removing all previously modules added and starting from scratch

transcripts. We used the Qualyzer[16] tool to code the transcripts and aggregate the codes.

The results of this qualitative analysis are divided into three sections. Section 5.1 investigates the programming barriers defined by Ko et al. [2004a] as they apply to Yahoo! Pipes. It then discusses activities that participants performed to overcome those barriers. Note that this section includes observations across both studies (Studies I and II) and both tasks. Section 5.2 explores reuse activities performed in Task 2 in both studies. Section 5.3 discusses debugging strategies used by study participants (Task 2 in Study II). Because Study I did not differentiate between EU and CSE participants, participants in this study are referred to as P[x], whereas participants in Study II are referred to as EU[x] or CSE[x].

5.1. Programming Barriers

Prior studies of end users creating mashups have led to the observation that users faced significant challenges [Cao et al. 2010b]. In our study, we wished to determine whether similar challenges existed in Yahoo! Pipes. Further, we wanted to understand whether knowledge of the provenance of a pipe and the ability to revert to an earlier successful version of a pipe would reduce some of the challenges.

5.1.1. Programming Barriers and Versioning Support. Ko et al. [2004a] identify six types of programming barriers that end users might face when programming in a new environment. Five of these have previously been identified by Cao et al. [2010b] to be present in the mashup domain (information barriers were not observed to be present). Table XI lists the five barriers, which include *understanding, use, coordination, selection, and design*; it also provides definitions and examples. We analyzed instances where participants faced problems using Yahoo! Pipes and classified each instance with the type of barrier (Table XII). We then investigated the occurrences of each barrier to gain insights into the challenges that participants faced and the strategies that they used to overcome the barriers. Our discussion of barriers is ordered based on the frequency with which these barriers were encountered.

There was a greater incidence of the first four barriers than of design barriers. This was an artifact of our task design. Design barriers occur primarily when users try to

[16]Qualyzer: http://qualyzer.bitbucket.org/#home.

Table XII. Instances of Learning Barriers with and without Versioning

Barriers	Instances of Learning Barriers	
	Ctrl	Exp
Understanding	135	84
Use	44	51
Coordination	37	16
Selection	31	26
Design	7	6
Total	**254**	**183**

formulate (design) a solution to a problem. Our tasks did not require participants to create a pipe from scratch; instead, our participants were given a pipe that they could reuse in Task 1 (Studies I and II) and a faulty pipe that they needed to debug in Task 2 (Study II). Because we saw few instances of design barriers, we do not discuss them further here.

Understanding barriers occur when a program's externally visible behavior obscures what the program does (or does not do) during execution. Yahoo! Pipes allows users to check the output of an entire pipe by clicking on the "output" module as well as the output of a single module individually. The debugger window found at the bottom of the editor interface (Figure 2) displays the results of the pipe (or the module). However, we found that the feedback provided was not self-explanatory. For example, participant EU12 in the Control group (Task 1) incorrectly used the URL for the `Fetch feed` module, and as a result, there was no output for any of the subsequent modules. However, there was no error information provided to notify him that the problem was in the input sources. EU12 ended up checking the output of each module (3–4 times) in the debugger window because of his inability to understand why there was no output from any modules. He ended up checking the output of modules for the entire pipe a total of 54 times but was not able to identify the error or its location.

A majority of participants had trouble understanding feedback when it was available. In fact, understanding barriers were the largest barrier faced by participants in both the Control and Experimental groups (135/254 and 84/183, see Table XII). Prior work has found understanding barriers to be the most difficult for users to overcome, calling them "insurmountable" [Ko et al. 2004b]. Therefore, it is pertinent to understand why these barriers occur and possible solutions.

We found that understanding barriers were reduced by 37.7% (from 135 instances to 84) in the presence of versioning support. There are two primary reasons for this reduction. First, participants who had versioning support needed to investigate fewer modules when they performed Task 2 in Study II, which required participants to debug two faulty modules. In the Control group, participants had no idea which modules were faulty and ended up having to check each module in the Pipe. In contrast, participants in the Experimental group were able to view the history of development and reduce the evaluation space; that is, they could identify the modules that were non-buggy (already checked and providing correct output) and focus on checking the rest. As a result, they investigated fewer modules and, therefore, faced fewer instances of understanding barriers. For example, although participant EU12, when working in the Experimental group, followed the same approach he had followed when working in the Control group (checking the output of each module that he needed to investigate), he needed to check the debugger window only 12 times instead of 54 times.

A second cause for the reduction in understanding barriers can be attributed to Task 1 in Study II, which required participants to comprehend the example pipe to be able to reuse parts of it in their tasks. On analyzing the participants' behavior, we found

that participants in the Experimental group faced fewer barriers because they could trace the evolution of the pipe and understand the functionality of each module and its aggregated effects on the pipe as each module was added (in the past). They could also note the output of the pipe or any selected module in prior (successful) versions. We posit that with the help of versioning, participants could comprehend the functionality of modules better and were better positioned to complete their "implementation" task.

Use barriers are caused by a lack of understanding of how to use a particular piece of code or module, and these posed the next largest challenge for participants (95/437, adding the use barrier instances across treatment groups in Table XII). For example, participant EU7 was frustrated when he tried to use the `loop` module to rename the titles of the feed. He spent 8.20 minutes experimenting with the `loop` module to make it work correctly, but was unsuccessful. We found that many participants had problems with this module, because it required a nested operation. That is, the operation that needs to be performed iteratively (`string replace`) had to be nested within the `loop` module. In general, participants generally had difficulty recognizing the functionality of modules in Yahoo! Pipes and the documentation/help provided by Yahoo! Pipes was inadequate.

We found that participants in the Experimental group actually exhibited a slightly greater incidence of this barrier (51 vs. 44 in Table XII). Versioning did not help participants because they faced use barriers when they tried to add new functionality, and the two modules that needed to be added were not part of the original pipe. Therefore the ability to view the history of development or having access to prior successful versions was not useful.

Coordination barriers were the next most prevalent type of barrier (53/437, adding relevant instances in Table XII). Such barriers are usually caused by a programming language's inherent limitations on how interfaces and individual programming units can be combined to perform complex functionalities [Ko et al. 2004a]. In our case, this problem occurred primarily when participants tried to (1) connect incompatible modules or (2) connect modules incorrectly.

We found coordination barriers to be more than halved in the presence of versioning support (16 vs. 37 instances, see Table XII). We found that coordination barriers were lower in the Experimental group because participants could view examples of how modules are supposed to be connected by viewing prior successful versions of the pipe. For example, participant EU4 in Task 2 tried to connect incompatible modules (the `number input` module which takes a "number" as input to the `truncate` module which takes "items" as input). He did not know why he was unable to connect them and commented, "Oh! It didn't connect ... this is interesting." He then went back to an earlier version of the pipe to view an example of how these modules were connected. After observing the connections in the older version, he was able to successfully connect the modules and complete his task correctly.

Selection barriers were the next most prevalent type of barrier faced by participants (57/437, see Table XII). These barriers are caused when a user is unable to identify the correct programming interface and how to achieve the correct behavior for a given program unit. In our situation, participants faced this problem primarily in the reuse task (Task 1, Studies I and II), where they needed to add new functionality that required the addition of two new modules. Many participants had difficulty selecting the correct Yahoo! Pipes module to implement the functionality. For example, in Task 1, participants were required to rename the titles of the search results to titles of their choice. To do so, they needed to use the operator `string replace` that replaces one string with another; however, many participants selected the wrong module (`rename`). In Yahoo! Pipes, the `rename` module is used to define mapping rules of different input formats (e.g., map input parameters to RSS formats). We found that participants made this mistake

because (1) the `rename` module appears before the `string replace` module in the list of available actions and (2) `string replace` is a computer science term that end users are not familiar with. For example, participant EU6 had this problem when he used the `rename` module and could not identify the correct module to use even after checking the help facility, which includes definitions of modules and examples of their use.

There was a small reduction in selection barriers when participants had versioning support. We found that this reduction was not attained when participants needed to add new functionality. Instead, the reduction occurred primarily in situations in which participants creating pipes began from scratch or incorrectly removed modules that they should have reused. In such cases, participants in the Control group faced selection barriers when they needed to figure out which modules to use, the actual behavior of those modules, and the correct usage of the modules. Participants in the Experimental group, in contrast, could refer to prior successful states of the given pipe to correctly use the module and therefore, faced fewer barriers. This is evident in cases in which participants in the Control group explicitly asked the observer how they could go back to a prior state. For example, participant EU6 asked the observer, "I cannot rename the titles can I go back to the first one I had. . .?"

5.1.2. Overcoming Barriers through Versioning Support. We found that versioning support helped participants overcome some barriers. Ko et al. [2004a] note that the programming barriers faced by users arise as a result of Norman's "gulf of execution" (the difference between users' intentions and actions available through the system) and "gulf of evaluation" (the effort to determine whether a desired goal has been achieved). Coordination and use barriers pose gulf of execution problems, understanding barriers pose gulf of evaluation problems, and selection barriers pose both gulf of execution and gulf of evaluation problems.

Here, we investigate how versioning support helped bridge the different gulfs and thereby alleviated barriers. Norman [1996] recommends bridging gulfs of execution by establishing visible constraints on the actions that are possible. Following this reasoning, coordination barriers can be reduced when implicit rules connecting modules have explicit representations. We believe that versioning support—and especially the ability to view prior versions—allowed participants to see how particular modules were connected, making the connection rules more explicit. For example, participant P6 commented, "It is easy to look at a partial part of the pipe in the versioning rather than looking at the entire pipe. It is easy to go back to the previous versions."

Similarly, use barriers can be alleviated if users can find examples of usage of difficult-to-use modules. For example, we have already noted that many participants had difficulty determining how to correctly use the `loop` module. Using versioning histories to provide examples of correct implementations can alleviate this problem.

To overcome gulfs of evaluation, Norman [1996] recommends that the current system state be accessible and understandable to users. While Yahoo! Pipes provides the ability to determine the output of a pipe at each module (clicking on the module lists the output in the debugger window), we found it to be insufficient. In fact, understanding barriers that were a result of gulf of evaluation problems posed the largest challenge for participants. While versioning support cannot directly help bridge the gulf of evaluation, the ability to view the provenance of a pipe and the output of each successful version helped to an extent. For example, a participant (EU10) noted, "I liked being able to see in a step-by-step manner in which it (the pipe) was created (which) made it easy to find the error."

Finally, selection barriers that pose both gulfs can be alleviated if users can view how complex functionalities can be implemented in Yahoo! Pipe modules. We did not include such a scenario in our study; participants faced the selection barriers when

Table XIII. Reuse Activities Adapted from Rosson and Carroll [1996]

Reuse Activities	Definition	Example
Finding a Usage Context	Finding the correct example pipe to reuse from	(none observed)
Evaluating a Usage Context	Assessing the appropriateness of the modules for reuse	A user selected (cloned) an appropriate version from the History of Pipe list as a starting point for creating his pipe
Debugging a Usage Context	Modifying the modules for reuse to fit task context	User began by executing the pipe to understand its functionality and modified the pipe to add additional functionality

they needed to implement new functionality by implementing two new modules. Still, we believe that versioning support can help users by mining different examples of implementations of such functionalities from the repository history.

5.2. Reuse

Reuse has been found to be a primary mechanism by which end users create new programs [Cypher et al. 2010]. In fact, a study of the Yahoo! Pipes repository that surveyed 32,887 pipes found that a majority of pipes (17,874%–54.35%) were cloned, of which a large portion (43%) were highly similar to each other [Stolee et al. 2011]. This suggests that finding an example pipe and reusing parts of the pipe is a fairly common occurrence in the Yahoo! Pipes community. However, reuse is no easy task, even for professional developers [Rosson and Carroll 1996]. Therefore, we wanted to investigate how end users perform reuse in Yahoo! Pipes and whether versioning support can help. Task 1 in both studies (Study I and II) required participants to reuse parts of a given pipe. Here we report on our observations of participants' reuse actions. We frame our observations on reuse (as originally defined) by Rosson and Carroll [1996] in their study of reuse in SmallTalk.

5.2.1. Reuse and Versioning Support. Rosson and Carroll [1996] identify three primary activities that users perform when they attempt reuse: *finding a usage context, evaluating a usage context, and debugging a usage context.* Table XIII presents the three reuse activities, their definitions within the context of our study, and an example of each.

Finding a usage context: Finding the correct usage context is the first step in reuse and relates to users locating the correct example with which to begin their reuse task. However, because in Yahoo! Pipes the majority of users create their pipes by cloning an existing pipe, our primary goal was to investigate how users understood the functionalities of the example pipe to create their own. In our study, we provided participants with an example pipe because we were not investigating how they identify the correct example.

Evaluating a usage context: The second step in reuse activities includes evaluating the appropriateness of the example to the task at hand. This activity includes executing the sample code and assessing the similarity of its functionality to the functionality required to complete the task. Similar to the study by Rosson and Carroll [1996], our task provided participants with a working (example) pipe that included components (modules) to be reused so that participants could easily execute the pipe to understand the functionality of the modules. Since the example pipe contained only a subset of modules that could be reused, participants needed to evaluate the context in which these modules were used and determine their appropriateness.

Participants used three options to evaluate usage context: (1) modify the example pipe by removing or adding modules, (2) create a clone of the pipe and then modify the clone, and (3) start a pipe from scratch. A majority of the participants began by modifying the given pipe directly, while a few participants followed the latter two strategies.

We found versioning support to be useful in helping participants evaluate the usage context of a module. For example, to evaluate the need for a particular module, participants in the Control group removed modules that they thought were irrelevant. However, many times they performed this step incorrectly. Because of this, participants needed to remember which modules they had removed and how to correctly connect those modules to the rest of the pipe. We observed that several participants struggled and were frustrated when they made erroneous decisions and wanted to revert to the original pipe to view it. In these cases, they viewed the original pipe by opening a new instance of the Yahoo! Pipes environment, viewing the example pipe and correcting their error. One participant, in fact, created a screen shot of the original pipe and referred to it when constructing the pipe. However, in the case of the Experimental group, participants could simply revert to a previous version (before they had removed the modules) or the original pipe through the History of Pipe list. For example, participant P3 commented, "Having versioning helps to see small building blocks makes it easy to see the entire picture. In the future, modifying/editing having all versions will be helpful."

Another instance in which versioning support was found to be useful involved cases in which participants viewed the provenance of the pipe to determine the correct point from which to begin their reuse work. That is, participants viewed the evolution history, found the version that had the majority of the functionality that was needed, executed that version to evaluate the appropriateness of the modules, and then cloned that version of the pipe for reuse. For example, participant P4 noted: *"You can see how each pipe was developed along the way to final product."* However, a side effect of providing information on the provenance of the pipe was that participants selected and executed every version in the history before they settled on the version from which to begin their reuse; this caused participants in the Experimental group to take extra time to complete their task.

Debugging a usage context: The final step in reuse includes users tweaking the reusable components to fit the context of their current task. In their study, Rosson and Carroll [1996] noted that upon finding an example component (in the Smalltalk library), programmers immediately moved into code development to try out that example. Their primary goal was to analyze the example component to understand how it would work for the new context and write new code by using the example component as a model. Once the component was plugged into the new context, they would analyze the workings of the reused component and any new code through testing. This method of development is termed "debugging into existence." We observed a similar trend with participants performing incremental development (one module at a time) and interleaving exploration of the modules in the given pipe with code development (bringing in a reusable module, modifying it, then testing it). The result was a highly contextualized, incremental analysis of the example application, and tightly integrated analysis and code development phases.

Participants in our study primarily analyzed Yahoo! Pipe modules by running the pipe and checking the output in the debugger window. When performing code development, however, they followed three distinct strategies: (1) exploring alternative ideas, (2) backtracking their changes, and (3) investigating prior successful states. We analyzed the study results to identify instances of each strategy (see Table XIV).

Table XIV. Instances of Explore Mechanisms Seen during Pipe Creation, with and without Versioning

Explore	Percentage of Instances of Explore Used	
	Ctrl	Exp
Alternative Ideas	80	38
Backtracking	74	36
Successful State	0	44
Total	**154**	**118**

Exploring Alternative Ideas. This strategy involves participants exploring different options to create a new functionality or remove an error. An example of this strategy was observed when participant CSE3 in Study II attempted to rename the news titles to his desired titles using the `string replace` module. When that did not provide the correct output, he explored using the `rename` module instead, so he removed the `string replace` module and connected the `loop` module with the `rename` module. However, that resulted in an error (incorrect use of `loop` module). He then tried different methods (spending 7.5 minutes) for connecting the two modules.

Participants in the Control group exhibited more instances of exploring alternative ideas (80 vs. 38, see Table XIV). This was primarily because participants in the Control group ended up reinvestigating some of the strategies that they had already attempted because they did not retain any explicit account of the strategies (the information resided in their heads). In contrast, participants in the Experimental group could identify the strategies that they had already investigated (modules added or removed) through the History of Pipe list.

Backtracking. We labeled activities as backtracking when participants explicitly reverted their changes because the line of development that they were pursuing was not fruitful. Participants backtracked primarily because of selection and use barriers. In our earlier example, when participant CSE3 realized that the `rename` module was incorrect, he backtracked through his changes to investigate the use of the `string replace` module again. He realized that the problem was not in the module but how it was connected. He commented, "Ummm, actually it was working with the string replace, now trying to figure out how to get that to work [in a loop]."

We found participants in the Control group to have greater instances of backtracking (74 vs. 36, see Table XIV). We found that participants in both groups made erroneous changes or changes that led to different results than what was expected. In such situations, they would revert their changes. The primary reason less backtracking was observed for the Experimental group was that participants better understood the functionality of modules in the given pipe and hence were able to correctly reuse the needed modules. They backtracked primarily when they were adding new modules (additional functionality). In contrast, participants in the Control group backtracked through their changes for additional modules as well as modules that were being reused.

Investigating Past Successful States. Since we recorded the version history of the given pipe as well as changes made by participants, they could execute any previous successful state in the development history. We identified instances in which participants in the Experimental group opened and executed a version of the example pipe (not including their changes) to understand how the development history of a pipe can aid in reuse. Such an example occurred when CSE2 (in Study II), while exploring how to use the `string replace` module in a `loop` module, remembered that the `loop` module was used in the sample pipe. He saved his current version, identified the prior

version that he wanted to view, and opened that version in the editor. He then tested that version to understand the workings of the particular module, and after he was satisfied, he simply reverted to his latest version to resume his work. In contrast, a participant in the Control group who wanted to view the modules in the original pipe would have to open another instance of the Yahoo! Pipes environment. If they needed to understand the effects of a particular module, they needed to manually isolate the workings of the modules of interest and then restore the original pipe—an error-prone and tedious activity.

5.2.2. Enhancing Reuse through Versioning. In our study, versioning support allowed participants to better understand the given pipe and the context in which modules of that pipe were used, which in turn helped them complete their reuse task with less backtracking or exploration of alternative ideas. However, versioning support was also useful when participants were adding extra functionality. We found that participants who had versioning support were more adventurous and explored more modules than those who did not, because they knew they could revert to an earlier state. For example, participant EU7, who was in the Control group after previously being in the Experimental group, created a set of changes while under the impression that the changes were being saved as versions. After a while, EU7 wanted to revert to an earlier stage of his pipe (program). However, because he was in the Control group, this was not possible through the interface, which led him to comment, "I don't have option for going back to an earlier version? So how should I know that it [what he has recently created] has an error?" There were other participants who explicitly requested versioning help while performing their reuse tasks. For example, when EU8 wanted to revert to his earlier changes, he asked, "Where is the undo window?"

5.3. Debugging

Debugging is an integral part of programming. Studies have shown that professional developers [Rosson and Carroll 1996] as well as students [Fitzgerald et al. 2010] spend significant portions of their time debugging. End users are no different. A study by Cao et al. [2010b] observed that end users creating mashups spent a significant portion of their time (76.3%) in debugging. These results indicate that debugging is not easy and should be better supported in programming environments. Since debugging is an essential activity, one of our goals was to observe the challenges that participants face when debugging in Yahoo! Pipes and how versioning support helps in debugging. We frame our observations by using the classification framework of debugging strategies first proposed by Grigoreanu et al. [2009], which was later used by Cao et al. [2010a] in their mashup study.

5.3.1. Debugging and Versioning Support. Table XV lists the different debugging strategies that participants used in our study and the definitions and examples of each strategy contextualized for our study. Table XVI displays the numbers of instances of each strategy observed in our study, with and without the use of versioning. We discuss the strategies that we observed in the order of their occurrence. In our ensuing discussion, we specify how versioning influenced debugging strategies.

Feedback following is defined as run time observation of program behavior. In Yahoo! Pipes, users can observe program behavior by viewing the output of an individual module or the entire pipe in the debugger window. However, we found that the feedback provided was insufficient. We have already noted that EU participants had trouble identifying problems when there was no feedback to follow. While versioning support cannot directly help users overcome this problem, it can be helpful in situations where the user can refer back to a correct example.

Table XV. Debugging Strategies Adapted from Grigoreanu et al. [2009] and Cao et al. [2010a]

Debugging Strategy	Definition	Example
Feedback Following	Investigating system generated feedback in the debugger window	User uses `translate` module to translate English feeds to Greek. He selects the `translate` module and observes the output in the debugger window.
Code Inspection	Inspecting modules in a pipe and pipe logic	User sequentially inspects each module of a pipe while understanding or debugging her pipe.
Testing	Testing a module by trying different input values	User supplies different "movie names" as input while running his pipe to test the correctness of the pipe.
Dataflow	Explicitly tracing data dependencies through a pipe	User had trouble understanding the top-down dataflow of the pipe as he tried to connect the "sort" module to the "union" module.
Proceed as in Prior Experience	Referring to a code snippet encountered in prior task or example pipe	While performing Task 2, user encounters `loop` module that he encountered earlier (in Task 1). He first consults usage of `loop` module in Task 1 before completing Task 2.

Table XVI. Debugging Strategies with and without Versioning

	Instances of Debugging Strategies	
Debugging	Ctrl	Exp
Feedback Following	463	218
Code Inspection	87	97
Testing	85	52
Dataflow	13	5
Proceed as Prior Exp.	22	6
Total	**680**	**378**

Code Inspection in Yahoo! Pipes involved participants scrutinizing the mashup logic, such as the functionality of modules, the parameter settings of modules, and the connections (wires) between modules. Participants traced the logic of each module by following the input and output of modules and their connections. Since modules in Yahoo! Pipes serve as black-box entities, participants attempted to understand the logic within a module by using help (tool tips or clicking the "?" icon) or checking the module's output. We found code inspection to be performed equally frequently across the treatment groups, the only difference being that participants in the Experimental group used the History of Pipe list to identify correct/tested versions and perform code inspection on those versions.

Testing is a critical step in debugging that involves isolating the parts of a program that could be faulty and then testing it to help identify the specific fault. In our studies, participants tried different approaches for doing this, but the most common involved isolating a module and testing it in isolation. A majority of participants used this approach in *Task2.movie* (aggregating movie reviews), which involved information from three different feeds combined together through a `union` module.

Participants in the Experimental group were able to reduce the amount of testing needed by identifying the modules that had already been tested from the History of Pipe list. While this helped participants identify the faulty modules, they did not

always know how to debug them, which accounts for the large number of testing steps required by EU participants. For example, participant EU11 was able to correctly a faulty module through the History of Pipe list, but unable to correct the fault. She remarked, "There is an error in truncate [module]. I do not know how to fix it."

Dataflow dependencies is primarily construct in Yahoo! Pipes because it is a data-driven environment. The spatial layout of the pipe typically reflects the dataflow, with modules laid out vertically and the output of a module (at a higher level) serving as an input to a module at a lower level. An incorrect understanding of the dataflow led to coordination barriers. Versioning support indirectly helped in these cases where past correct examples could be viewed. Still, some participants had trouble understanding the dataflow in the given pipes, and versioning support was not helpful when participants had issues understanding the dataflow concept. Providing examples of how specific modules are connected could alleviate this; dataflow problems are dynamic and depend on inputs and cannot be alleviated by simply viewing a static picture of a pipe. Rather, a dynamic view of the data processing and flow needs to be made explicit.

Proceed as in prior experience is a strategy in which participants refer to their prior experiences in building a similar application to help them in their current task. In our study the versioning history served as an external experience repository. Some participants viewed their earlier tasks or the given (earlier) sample pipes to complete their current task. For example, when participant EU4 in his second task was trying to connect two incompatible modules and was unable to do so, he remembered that he had used similar modules in his earlier task and opened the earlier pipes that he had completed. After checking the structure and connection logic in one of the examples, he was able to correctly connect the modules in his current pipe.

5.3.2. Enhancing Debugging through Versioning. The greatest debugging assistance provided by versioning was allowing participants to reduce the testing space. That is, participants in the Experimental group could easily identify the parts of the pipes that had already been tested and were therefore correct, so they needed to debug only parts of pipes. While versioning cannot directly help users debug a pipe, it can provide examples of correct usage of modules that can serve as a reference. Users can test each module as it was added during development to identify problem spots and focus their debugging efforts on those spots.

5.4. Summary

Here, we summarize our qualitative results with respect to how versioning support helped participants overcome learning barriers, reuse parts of pipes, and perform debugging.

We found that our participants faced several learning barriers. *Understanding* barriers were the most prominent, followed by *use*, *coordination*, and *selection* barriers. We also observed that end users faced these learning barriers to a greater degree than their more experienced (CSE) counterparts.

We found that versioning helped reduce *understanding* barriers because participants could view the provenance of their pipes and selectively execute prior versions, which helped them comprehend the functionality of the pipes in an incremental fashion. *Use, coordination*, and *selection* barriers, which all deal with correctly identifying the appropriate module to use and connecting it correctly, did not receive direct help from versioning support. However, versioning support did help reduce these barriers when participants could find previous examples of modules, the manner in which they were used (e.g., parameters used), and their connections (e.g., wiring between them).

In the case of reuse, we found that versioning support allowed participants to correctly comprehend the functionality of pipes by observing how they evolved (using the

History of Pipe list). Furthermore, we found that in the Experimental group, where versions were automatically created, participants were able to revert back from mistakes and as a result were less risk averse in their explorations. Another indirect benefit of versioning was that strategies that had already been employed (and their results) were explicitly visible in relation to pipes; this led participants to perform less backtracking and to avoid re-implementing failed strategies.

In the case of debugging, participants largely followed code inspection and testing strategies. In these cases, version histories for pipes as well as results pertaining to executions of versions (erroneous or tested) helped reduce the evaluation space for participants. That is, participants needed to check fewer modules, because some had already been tested.

6. DISCUSSION AND IMPLICATIONS

In this section, we provide a general discussion of our study and results and then suggest several further implications.

6.1. Discussion

As we have noted, web mashups are an important class of software system, popular among end users and with support via several different mashup programming environments, most of which provide visual interfaces. As noted by others, and as reflected in the findings presented in this article, end-user programming mashups tend to employ opportunistic programming, tend to rely heavily on reuse of existing mashups, and can often be seen to debug their programs into existence. End users currently engage in these behaviors, however, in an ad-hoc manner, and mashup environments do not necessarily support these activities to the extent they could. Moreover, mashup programmers face several programming barriers, including understanding, use, coordination, and selection barriers.

In this work, we studied the use of versioning support in mashup programming environments, focusing on the Yahoo! Pipes platform, to determine whether that support can help end users in their programming tasks. Our results reveal that, indeed, versioning support can be useful to both end user and more sophisticated programmers working in the Yahoo! Pipes context. In particular, versioning support can help with the very reuse and debugging activities that these programmers engage in and can alleviate some of the programming barriers that they face.

To provide versioning support, we studied the features of versioning systems available to professional programmers. To better conceptualize versioning support in the Yahoo! Pipes environment itself, we considered various user scenarios. We mapped various tasks performed by users to versioning system concepts. Based on these mappings, we prototyped and implemented the versioning system for Yahoo! Pipes using a visual interface. Automating support for versioning was important in this context, as it allowed us to retain versions and present information about versions to users without requiring them to learn formal versioning system concepts.

The "History of Pipe" list is a primary component of our versioning support, allowing users to view versions, differences between them, and status on their operational status. We chose to show modules as the basic variation unit for displaying differences between pipes, in part because the average size of pipes has been shown to be relatively small, in the range of six to eight modules [Stolee et al. 2011]. This notion could be extended, however, to include other components of variation such as connectors or parameters.

Web mashup creation involves collecting data, manipulating data, and building interfaces [Zang and Rosson 2008]. We designed our experiment tasks to allow us to observe the last two of these activities, as these require problem solving and creativity.

We also used common data sources such as news, blogs, shopping, and movies in our tasks. We used sample pipes from the Yahoo! Pipes repository and created versions of those pipes. We selected modules to include in versions based on our understanding of the functionality found in different variations of pipes (generated by us). For our experimental group, there were six versions for Task 1 and seven versions for Task 2, along with their execution histories. These assumptions may not reflect a variation space in real life, because users may tend to create far more variations.

The primary lessons derived from our studies relate to programming strategies, debugging strategies, and the use of examples.

The presence of versioning helped our participants in programming by allowing them to explore various variations of pipes. Participants with versioning support were less risk averse while creating mashups, and this helped them experiment with and choose between different ideas, aided by the ability to return to prior successful and unsuccessful changes. This approach fits nicely with the processes of opportunistic programming and programming by successive refinement.

Our study participants also debugged mashups through a process that included successive refinements. Providing versioning support helped the participants do this, because they could look for prior successful strategies and avoid prior unsuccessful strategies. Further, the "History of Pipe" list helped participants focus their debugging efforts, effectively reducing the evaluation space they needed to explore.

Finally, programmers (end users and nonalike) are known to learn from examples [Lieberman et al. 2006], and our study participants were no exception in this respect. With versioning support, our participants were able to rely not only on examples found in the repository, but also on prior versions of pipes. When participants had difficulties understanding how a module worked or needed to be connected with other modules, they preferred to look at older versions in which that module was used and implement similar features in their current pipe.

6.2. Implications

Our qualitative analysis prompts us to suggest several implications for ways in which mashup programming environments can be improved; these involve providing better feedback, support for visualizations, testing and debugging support, and recommendation systems.

6.2.1. Providing Better Feedback. We found that a large number of the programming barriers faced by our study participants were caused because of poor feedback and help functionality provided by Yahoo! Pipes. The largest barriers (understanding) arose because participants could not understand the runtime feedback or error messages. Providing explicit feedback in a language that is accessible to end users can reduce such understanding barriers. Improved documentation will also help with understanding, use, and selection barriers. Similarly, built-in examples may help users understand the general usage of modules and reduce both use and coordination barriers. Improved tool tip help functionality could provide quicker access to documentation, as we found participants to overwhelmingly prefer help within the environment to help found outside.

Better feedback in the form of visual cues that build on intuitive color and visual concepts could also help users be more effective in creating wire-oriented mashups. For example, we observed that our study participants tended to follow the color interpretations that we use in daily life (e.g., green depicts go, red means error/danger), and these are not consistently used in Yahoo! Pipes. For example, Yahoo! Pipes highlights the module that is under inspection (or being debugged) in orange, and some participants had difficulty determining whether this color should be seen as an error

or a warning sign. For instance, one of the EU participants asked, "Why is the module orange? Does it mean I have an error?"

6.2.2. Supporting Exploration through Better Visualizations. End-user activities like opportunistic programming, debugging into existence and reuse tend to create large numbers of variations of programs. We have supported users' explorations among variations through the "History of Pipe" list. Better visualizations are needed, however, to facilitate exploration.

For example, our versioning interface could be enhanced to provide more intuitive views of versioning history and differences. Current source code revision tools provide editors that allow users to compare source files side by side [Heckel 1978]. We could implement a similar editor in which versions of pipes could be shown side by side, with differences highlighted. Furthermore, we currently display a linear history of the evolution of pipes; this could be improved to show relationships in a graphical manner.

There is also a need to identify visual features and appropriate levels of abstraction in the end-user environment to help users leverage information on variations. Using an appropriate level of visualization will help users explore variations or view all variants at once. For example, we expect that the appropriate level of abstraction for a user while viewing variants in their current work space will be a fine-grained view (similar to that of the History of Pipe list, while a coarse-grained view (e.g., family tree) may be better while viewing all variants of a program in the repository.

6.2.3. Supporting Testing and Debugging. There is a need for better testing techniques and better debugging tools to help programmers create dependable mashups. Grammel and Storey [2008] have stressed the need for such techniques, and our observations underscore this need. In our studies, participants had difficulty locating the sources of errors and engaged in testing activities in an ad-hoc manner. Support for more rigorous testing and debugging methodologies could help users attain more dependable mashups. Support for better error reporting could also help users assess and correct problems that occur. Testing and debugging techniques created to assist end-user programmers such as "Whyline" [Ko and Myers 2004] and WYSIWYT [Fisher et al. 2006] may be of further help. Finally, once pipes have been constructed and made available to the community, the community itself may be of help. Social recommendation systems like *"HelpMeOut"* that help users debug error messages by suggesting solutions that peers have applied in the past are known to be helpful [Hartmann et al. 2010b].

6.2.4. Creating Recommendation Systems for Finding and Using Artifacts. Learning by examples has been shown to be an effective technique for end users [Lieberman et al. 2006]. A large percentage of programs in Yahoo! Pipes are clones (or pipes that have been reused) [Stolee et al. 2011]; thus, recommendation systems that identify syntactically and semantically similar pipes can help end users in their development efforts. Keeping histories and versions in pipe repositories will assist the creation of recommendation systems. Information on usage patterns and the evolution of mashups can be used to recommend more fine-grained mashup components than can currently be identified by users. Furthermore, because version histories are capable of capturing the successful and unsuccessful states of mashups, more stable and correct versions can be recommended. Finally, social recommendation systems that can connect novices with the creators of example pipes can allow users to learn from the community.

We also believe that recommendation systems that guide users in their attempts to use modules in Yahoo! Pipes will help alleviate learning barriers. By considering modules in terms of their structures and parameters, such recommendation systems should be able to suggest not only entire pipes as sources for information but also

modules and parameters. In summary, recommendation systems should be able to connect users with the information they need to build dependable pipes more efficiently and effectively.

7. RELATED WORK

Here we discuss related work on end users and mashups, followed by work on versioning, histories, and visualizations.

7.1. End Users and Mashups

There has been quite a bit of recent research related to mashups. Zang and Rosson [2008] investigate the types of information that users encounter and the interactions between information sources that they find useful in relation to mashups. The authors also examine data gathering and integration [Zang and Rosson 2009] and discuss results of a study of web users, focusing on their perceptions of what mashups could do for them and how they might be created. They also note that, when asked about the mashup creation process, end users could not even describe them in terms of the three basic steps of collecting, transforming, and displaying data.

There has been recent research aimed at understanding the programming practices and hurdles that mashup programmers face in mashup building. Cao et al. [2010b] discuss problem solving attempts and barriers that end users face while working with mashup environments, and describe a "design-lens methodology" to view programming. Cao et al. [2010a] also study a debugging perspective on end-user mashup programming.

Researchers have also empirically studied the Yahoo! Pipes environment itself in order to understand the programming practices and issues faced by end users and their communities. Jones and Churchill [2009] describe various issues faced by end users while developing web mashups using the Yahoo! Pipes environment. They observe the conversations of the users in discussion forums in order to understand the practices followed, problem solving tactics employed, and collaborative debugging engaged in by these online communities of end users.

Dinmore and Boylls [2010] empirically studied end-user programming behaviors in the Yahoo! Pipes environment. They observe that most users sample only a small fraction of the available design space, and simple models describe their composition behaviors. They also find that users attempt to minimize the degrees of freedom associated with a composition as it is built and used.

Stolee et al. [2011] analyzed a large set of pipes to understand the trends and behaviors of end users and their communities. They observe that end users employ the repository in different ways than professionals, do not effectively reuse existing programs, and are not aware of the community. In another study [Stolee and Elbaum 2011], they classified various "smells" (programming errors) found in Yahoo! Pipes and devised refactoring techniques that can be used to remove those smells from the pipes.

7.2. Versioning, Histories, and Visualization

Versioning has been the subject of substantial prior work. Versioning capabilities are heavily used in commercial software development and are required for a team to be successful. Versioning is used by professional developers to keep track of changes (theirs and others), share or benchmark the latest versions of their code, or revert their changes [Tichy 1985]. Tools such as diffIE have been used to keep track of changes in webpages [Teevan et al. 2009, 2010]. Our versioning capabilities also allow end users to keep track of changes and revert their changes. In addition, we allow users to benchmark significant events as an aid in debugging mashups.

History mechanisms such as undo or "time travel" enable revisitation of earlier versions in a variety of applications [Berlage 1994; Derthick and Roth 2001; Edwards et al. 2000; Meng et al. 1998]. History tools can play an important part in visualization processes, supporting iterative analysis by enabling users to review, retrieve, and revisit visualization states. Graph visualizations can help to present, manage, and export histories [Eric et al. 1995; Heer et al. 2008; Hightower et al. 1998; Kaasten and Greenberg 2001; Klemmer et al. 2002].

There has been some work in revision control on visual interfaces specifically meant for capturing design histories. This includes work related to revision histories and determining differences between UML diagrams [Chen et al. 2003; Ohst et al. 2003], CASE diagrams [Mehra et al. 2005], and statecharts [Schipper et al. 2009]. To facilitate design, some tools to track and visualize changes exist [Hartmann et al. 2010a]. Our work differs from these in two aspects: (1) these revision control mechanisms are used for designing software or visual media (WYSIWYG document editors, movie producers, and video game developers), whereas our work targets mashups; and (2) these mechanisms are built for designers, whereas our target population is end users.

Some end-user environments such as Google Docs and Google Websites provide basic versioning facilities to enable group editing, but these capabilities are only for text edits. These environments also allow versions to be created on each save. In these environments, however, the versions are represented only with version numbers of files. In contrast, our versioning support is more fine-grained, providing a list-view of each pipe that helps end users view pipe constructs (module names) in the order in which they were added to the pipe canvas.

8. CONCLUSION

In this article, we presented our Pipes Plumber extension to Yahoo! Pipes, which provides versioning support for mashup programmers using that environment. Our empirical results studying the use of that environment in mashup creation and debugging tasks provide evidence that our versioning support can help mashup programmers create and debug mashups, and this includes both individuals who have formal programming experience and those who do not. Our qualitative analysis reveals additional insights into the ways in which versioning addresses barriers faced by mashup programmers, reuse problems, and problems in debugging.

It would be useful to extend our environment to provide further visualization support to mashup programmers. In particular, methods for presenting version histories in forms such as family tree–like structures may provide an appealing metaphor for end users. As discussed in Section 7, various visual representations for viewing the histories of graphs or documents have been suggested in prior work (e.g., [Eric et al. 1995; Heer et al. 2008; Hightower et al. 1998; Kaasten and Greenberg 2001; Klemmer et al. 2002; Woodruff et al. 2001]). There has also been work in the web domain on methods for keeping histories of the webpages visited and visual ways for depicting these histories [Jatowt et al. 2008]. This prior work may provide useful mechanisms for use in mashup programming environments. It would also be useful to conduct additional studies of our versioning approach, in particular using larger pipe artifacts, and considering longer-term scenarios in which participants construct pipes and versions over extended periods of time.

ACKNOWLEDGMENTS

We thank Branden Barber and Amanda Swearngin for helping transcribe recording sessions of the participants and helping with the grading task. We thank Margaret Burnett for her feedback. We thank the

anonymous reviewers and the associate editor for comments and suggestions that substantially improved the article. We also thank our study participants.

REFERENCES

Thomas Berlage. 1994. A selective undo mechanism for graphical user interfaces based on command objects. *Transactions on Computer Human Interaction* 1, 3 (September 1994), 269–294.

Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Opportunistic programming: Writing code to prototype, ideate, and discover. *IEEE Software* 26, 5 (September 2009), 18–24.

Jill Cao, Kyle Rector, Thomas H. Park, Scott D. Fleming, Margaret Burnett, and Susan Wiedenbeck. 2010a. A debugging perspective on end-user mashup programming. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. 149–156.

Jill Cao, Yann Riche, Susan Wiedenbeck, Margaret Burnett, and Valentina Grigoreanu. 2010b. End-user mashup programming: Through the design lens. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 1009–1018.

Ping Chen, Matt Critchlow, Akash Garg, Chris Van Der Westhuizen, and André Van Der Hoek. 2003. *Software Product-Family Engineering*. Springer Verlag, 269–281.

Ron Cody. 1988. *SAS Workbook*. Vol. 1. SAS Publishing.

Allen Cypher, Mira Dontcheva, Tessa Lau, and Jeffrey Nichols. 2010. *No Code Required: Giving Users Tools to Transform the Web*. Morgan Kaufmann.

Mark Derthick and Steven F. Roth. 2001. Enhancing data exploration with a branching history of user operations. In *Knowledge Based Systems*. 65–74.

Matthew D. Dinmore and C. Curtis Boylls. 2010. Empirically-observed end-user programming behaviors in Yahoo! Pipes. In *Psychology of Programming Interest Group*.

Shirley Dowdy, Stanley Wearden, and Daniel Chilko. 2004. *Statistics for Research*, 3rd ed. Wiley.

W. Keith Edwards, Takeo Igarashi, Anthony LaMarca, and Elizabeth D. Mynatt. 2000. A temporal model for multi-level undo and redo. In *Proceedings of the ACM Symposium on User Interface Software and Technology*. 31–40.

May Eric, Eric Z. Ayers, and John T. Stasko. 1995. Using graphic history in browsing the world wide web. In *International World Wide Web Conference*. 11–14.

Marc Fisher, Gregg Rothermel, Darren Brown, Mingming Cao, Curtis R. Cook, and Margaret Burnett. 2006. Integrating automated test generation into the WYSIWYT spreadsheet testing methodology. *ACM Transactions on Software Engineering and Methodology* 15 (April 2006), 150–194.

Sue Fitzgerald, Renée McCauley, Brian Hanks, Laurie Murphy, Beth Simon, and Carol Zander. 2010. Debugging from the student perspective. *Transactions on Education* 53, 3 (April 2010), 390–396.

Lars Grammel and Margaret-Anne Storey. 2008. *An End User Perspective on Mashup Makers*. Technical Report DCS-324-IR. Department of Computer Science, University of Victoria.

Valentina Grigoreanu, James Brundage, Eric Bahna, Margaret M. Burnett, Paul Elrif, and Jeffrey Snover. 2009. Males' and Females' Script Debugging Strategies. In *Proceedings of the International Symposium on End-User Development*. 205–224.

Björn Hartmann, Sean Follmer, Antonio Ricciardi, Timothy Cardenas, and Scott R. Klemmer. 2010a. d.note: revising user interfaces through change tracking, annotations, and alternatives. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 493–502.

Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010b. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 1019–1028.

Paul Heckel. 1978. A technique for isolating differences between files. *Commun. ACM* 21, 4 (April 1978), 264–268.

Jeffrey Heer, Jock D. Mackinlay, Chris Stolte, and Maneesh Agrawala. 2008. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *Transactions on Visualization and Computer Graphics* 14 (November 2008), 1189–1196.

Ron R. Hightower, Laura T. Ring, Jonathan I. Helfman, Benjamin B. Bederson, and James D. Hollan. 1998. PadPrints: Graphical multiscale web histories. In *Proceedings of the ACM Symposium on User Interface Software and Technology*. 58–65.

Angus F. M. Huang, Shin Bo Huang, Evan Y. F. Lee, and Stephen J. H. Yang. 2008. Improving end-user programming with situational mashups in Web 2.0 environments. In *IEEE International Symposium on Service-Oriented System Engineering*. 62–67.

Adam Jatowt, Yukiko Kawai, Hiroaki Ohshima, and Katsumi Tanaka. 2008. What can history tell us?: Towards different models of interaction with document histories. In *Proceedings of the Hypertext and Hypermedia*. 5–14.

Michael Jones and Christopher Scaffidi. 2011. Obstacles and opportunities with using visual and domain-specific languages in scientific programming. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. 9–16.

M. Cameron Jones and Elizabeth F. Churchill. 2009. Conversations in developer communities: A preliminary analysis of the Yahoo! Pipes community. In *Proceedings of the International Conference on Communities and Technologies*. 51–60.

Shaun Kaasten and Saul Greenberg. 2001. Integrating back, history and bookmarks in web browsers. In *Extended Abstracts on Human Factors in Computing Systems*. 379–380.

Scott R. Klemmer, Michael Thomsen, Ethan Phelps-Goodman, Robert Lee, and James A. Landay. 2002. Where do web sites come from?: capturing and interacting with design history. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 1–8.

Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The state of the art in end-user software engineering. *Comput. Surveys* 43, 3 (April 2011), 21:1–21:44.

Andrew J. Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 151–158.

Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. 2004a. Six learning barriers in end-user programming systems. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. 199–206.

Andrew J. Ko, Brad A. Myers, and Htet H. Aung. 2004b. Six learning barriers in end-user programming systems. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. 199–206.

Sandeep Kaur Kuttal, Anita Sarma, and Gregg Rothermel. 2011a. History repeats itself more easily when you log it: Versioning for mashups. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. 69–72.

Sandeep Kaur Kuttal, Anita Sarma, Amanda Swearngin, and Gregg Rothermel. 2011b. Versioning for mashups—An exploratory study. In *Proceedings of the International Symposium on End-User Development*. 25–41.

Clayton H. Lewis. 1982. *Using the "Thinking Aloud" Method In Cognitive Interface Design*. RC 9265. IBM.

Henry Lieberman, Fabio Paterno, and Volker Wulf. 2006. *End User Development*. Vol. 9. Springer Netherlands.

Akhil Mehra, John Grundy, and John Hosking. 2005. A generic approach to supporting diagram differencing and merging for collaborative design. In *Proceedings of the International Conference on Automated Software Engineering*. 204–213.

Chii Meng, Motohiro Yasue, Atsumi Imamiya, and Xiaoyang Mao. 1998. Visualizing Histories for Selective Undo and Redo. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 459–464.

Don Norman. 1996. *The Psychology of Everyday Things*. Vol. 1. Basic Books.

Dirk Ohst, Michael Welle, and Udo Kelter. 2003. Differences between versions of UML diagrams. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 227–236.

Mary Beth Rosson and John M. Carroll. 1993. Active programming strategies in reuse. In *Proceedings of the 7th European Conference on Object-Oriented Programming*. Springer-Verlag, London, UK, 4–20.

Mary Beth Rosson and John M. Carroll. 1996. The reuse of uses in Smalltalk programming. *Transactions on Computer Human Interaction* 3, 3 (September 1996), 219–253.

Christopher Scaffidi, Mary Shaw, and Brad A. Myers. 2005. Estimating the numbers of end users and end user programmers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. 207–214.

Arne Schipper, Hauke Fuhrmann, and Reinhard von Hanxleden. 2009. Visual Comparison of Graphical Models. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*. 335–340.

Katie Stolee, Sebastian Elbaum, and Anita Sarma. 2011. End-User programmers and their communities: An artifact-based analysis. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. 147–156.

Kathryn T. Stolee and Sebastian Elbaum. 2011. Refactoring pipe-like mashups for end-user programmers. In *Proceedings of the International Conference on Software Engineering*. 81–90.

Jaime Teevan, Susan T. Dumais, and Daniel J. Liebling. 2010. A longitudinal study of how highlighting web content change affects people's web interactions. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 1353–1356.

Jaime Teevan, Susan T. Dumais, Daniel J. Liebling, and Richard L. Hughes. 2009. Changing how people view changes on the web. In *Proceedings of the ACM Symposium on User Interface Software and Technology*. 237–246.

Walter F. Tichy. 1985. RCS-A system for version control. *Software: Practice & Experience* 15, 7 (July 1985), 637–654.

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2000. *Experimentation in Software Engineering: An Introduction*. Springer.

Allison Woodruff, Andrew Faulring, Ruth Rosenholtz, Julie Morrsion, and Peter Pirolli. 2001. Using thumbnails to search the Web. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 198–205.

Nan Zang and Mary B. Rosson. 2008. What's in a mashup? And why? Studying the perceptions of web-active end users. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. 31–38.

Nan Zang and Mary B. Rosson. 2009. Playing with information: How end users think about and integrate dynamic data. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. 85–92.

Received October 2013; revised December 2013; accepted December 2013