Commit Bubbles

Titus Barik*†, Kevin Lubick†, Emerson Murphy-Hill†
*ABB Corporate Research, Raleigh, USA
†North Carolina State University, Raleigh, USA
titus.barik@us.abb.com, kjlubick@ncsu.edu, emerson@csc.ncsu.edu

***Abstract*—Developers who use version control are expected to produce systematic commit histories that show well-defined steps with logical forward progress. Existing version control tools assume that developers also write code systematically. Unfortunately, the process by which developers write source code is often evolutionary, or as-needed, rather than systematic. Our contribution is a fragment-oriented concept called Commit Bubbles that will allow developers to construct systematic commit histories that adhere to version control best practices with less cognitive effort, and in a way that integrates with their as-needed coding workflows.**

I. Motivation

In version control systems, a *commit* represents an atomic set of changes with respect to a previous state; together, these sequences of commits form a *commit history* [1]. There are many best practices when adding commits to version control commit histories. For the commit itself, these best practices include using a descriptive commit message; avoiding *indiscriminate* commits, that is, commits that blindly include all changed files; and making each commit a "logical unit" — such that each commit has a singular purpose [2]. Extending this idea, the resulting *published* commit histories should, in some sense, be *systematic*, in that the history shows well-defined steps, each with logical forward progress, telling a cohesive narrative without broken or suboptimal steps [1].

Yet the way in which developers write and edit source code is not commonly done in a *systematic* way, but an *as-needed* way instead [3], [4]. When using a systematic strategy, developers first construct a plan to complete a set of tasks and only then make the edits (e.g., waterfall). In contrast, when adopting an as-needed strategy, developers identify a relevant point in the program and continue making edits until the solution emerges (e.g., agile).

The fundamental problem is that the as-needed strategy developers frequently use to write code is incompatible with the systematic strategy that developers would need to use in order to generate their published commit histories. As evidence, Murphy-Hill and colleagues found that refactoring operations are performed frequently, and that programmers frequently interleave refactoring with other types of programming activity [5]. Similarly, Negara and colleagues found that 46% of refactored program entities are interspersed with other changes [6], and Kusunoki and colleagues found that 20%-60% of commits intersperse peripheral changes, such as code formatting [7]. In version control histories, these as-needed edits manifest themselves as *tangled* commits, that is, commits that contain two or more logical units of changes [8], and as incomplete or incorrect commit messages, which fail to capture the full description of the change [9], [5]. Both of these issues are obstacles to supporting downstream change management tasks, such as merging commits between branches, and conducting effective code reviews [8].

Parnas and Clements propose that although real software is rarely developed systematically, it should be possible to make it appear that software was designed by such a process [10]. In version control, for instance, one mechanism for accomplishing this is to untangle commit histories into systematic histories using history revision operations offered by version control systems.[1] For example, a developer can theoretically reorder or delete commits using *rebase*. However, in practice, this procedure is difficult to perform correctly [11].

In this paper, we argue that the primary barrier to performing effective history revision is that current revision tools inadequately align their tool functionality with developer workflows [12]. First, revision activities, if they occur at all, are typically performed as a distinct activity from coding [3]. This context switch makes it difficult to remember all of the change activities related to a particular commit, since human memory is particularly failure-prone during these switches [13]. Second, history revision as supported in tools today takes the perspective that developers are ordinarily able to successfully create systematic histories, and that revision is an exceptional situation. However, research indicates that exceptions are normal in work processes, and tools should support handling exceptional situations as routine [14].

Our vision is a development model that reconciles as-needed coding activities with systematic commit activities. We operationalize this model by adding commit support to *Code Bubbles*, a metaphor and tool that allows developers to reason in terms of fragments and working sets, and allows for fluid rearrangement and manipulation of these working sets [15]. Our proposed extension, *Commit Bubbles*, supports developers by a) blending coding and commit activities through fragments to minimize context switching, and b) treating history revision as a routine, rather than exceptional process. Our contribution is a concept that will allow developers to construct commit histories that adhere to version control best practices with less cognitive effort, and in a way that integrates with their as-needed coding workflows.

[1] Robertson calls this "sausage making" — "The process of developing software, similar to the process of making sausage, is a messy, messy business [. . .] If you hide the sausage making, you can create a beautiful looking history where each step looks as delicious as the end-product" [11].

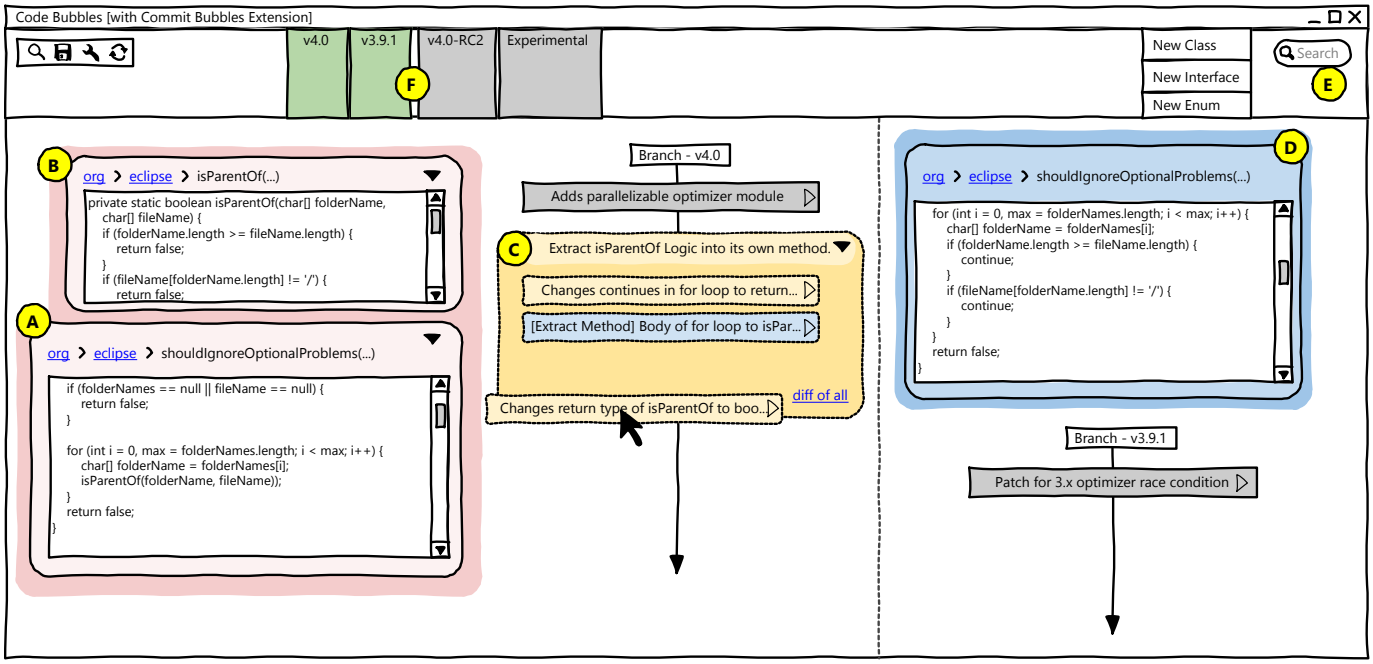

Fig. 1. A mockup of Code Bubbles, extended with Commit Bubble elements. (A), (B) and (D) are code bubbles, which can be placed on the screen by using the search bar (E) or a dragging-and-dropping from a navigation tree (not shown). (F) shows a task context, that is, a panel of working sets. A commit bubble can be expanded (C) to reveal additional information. In this figure, (C) consists of two squashed commits, with a third commit bubble being added to this set using drag-and-drop. The environment offers an infinitely scrollable canvas where the developer performs both coding and commit activities.

II. Commit Bubbles

Researchers recognize the cognitive benefits of tools that support thinking in terms of arbitrary subsets of source code, or *fragments*, rather than files [15], [16], [17], [18]. Fragments offer a metaphor for displaying relevant pieces of information in an as-needed way, for example, when coding or debugging. We postulate that version control activities frequently require fragment-based thinking, for example, when comparing code at two different states, when determining an atomic set of changes to commit, and when reordering commit histories.

We chose Code Bubbles to prototype our concept because it is an open source, fragment-based development environment. Code Bubbles realizes the metaphor of light-weight editable code fragments as *bubbles* (Figure 1). The bubbles metaphor makes it easier for developers to see many fragments of information at once, without having to context switch between different windows or tools. The metaphor also enables fluid manipulation of fragments without enforcing rigid boundaries about where information should be placed.

Our contribution, Commit Bubbles, extends fragment-based tools to support the manipulation of commits, an essential activity to translating as-needed activities into a systematic history. Existing version control tools require that developers code in a systematic way. Therefore, these tools assume that commit histories by default align with version control best practices, and that revision is rare. Our insight is that by treating revision as a frequent and routine activity, developers will be able to continue working in their as-needed way, while simultaneously producing systematic histories.

In the next section, we narrate a user experience that demonstrates how a developer would use Commit Bubbles with existing fragment-based environments in their daily programming activities to correct a defect present in multiple versions of the code base.

Example User Experience

Anthony is a software developer who has been tasked with adding a new feature to the Eclipse JDT 4.0 branch.[2] During this coding activity, he opens the method `shouldIgnoreOptionalProblems` as a code bubble (Figure 1A). As he examines the method, he notices a particularly complicated piece of logic in the loop, which he decides to refactor to `isParentOf` using a combination of the extract method tool of his integrated development environment and some manual cleanup (Figure 1B). Through a change detection algorithm, Commit Bubbles generates three new commit bubbles to reflect these actions (Figure 2).

These three commit bubbles are all part of the same change, so he *squashes*, that is, combines them into one larger commit bubble by dragging and dropping. He edits the autogenerated commit message (Figure 1C) to better reflect his intent, by clicking and then typing the new message.

Anthony is about to handle the return value of the newly extracted method when he notices the extracted code has a bug — `isParentOf` only checks for the Unix line separator

[2]This example is based on the `Main.java` file located at http://download.eclipse.org/eclipse/downloads/drops4/R-4.4.1-201409250400/#JDTCORE.

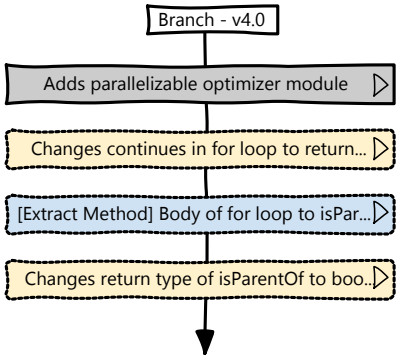

Fig. 2. Commit bubbles are generated automatically while developing code. A solid outline indicates the commit bubble has been stored to public history, and dashed outlines indicate commits that can be reordered. These autogenerated commit bubbles give a starting point for a systematic history.

('/'). Anthony adds a condition for the Windows line separator ('\\') and creates a local variable to reduce redundant code. As before, he squashes these two commit bubbles into one (Figure 3A). This bug exists in the 3.9.1 branch as well, which he verifies by instantiating the corresponding code bubble (Figure 1D). With Commit Bubbles, Anthony can have more than one version of a fragment open at time.

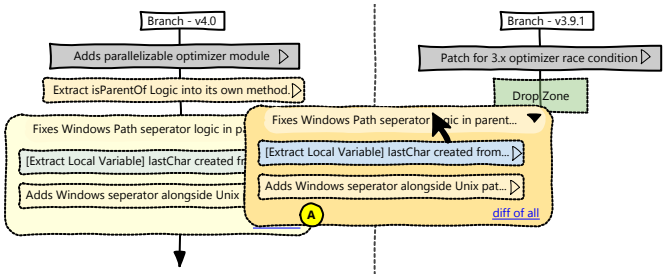

Fig. 3. Commit bubbles are copied from one branch and integrated into another by dragging and-dropping the commit bubble from the source branch to the target branch. While existing tools optimize for making commits, our tool makes revising history tasks just as easy.

In existing version control tools, Anthony would not be able to easily transfer his commits across the two branches; line-based patches do not understand the semantics of the code and fails to apply the patch. He would not be able to transfer the refactoring commits (Figure 1C) and the bugfix either because the branch is in a strict "bugfixes only" cycle which does not allow non-essential changes such as refactorings. Decoupling this relatively simply refactoring change from the bugfix is unexpectedly nontrivial. One recommended "low-tech" way to do this would be to "save all your edits, then re-introduce them in logical chunks, committing as you go" [2]. More sophisticated approaches exist, but they are even more burdensome and error-prone because they rely on complex and non-obvious data structure-level manipulations [11]. Worst of all, in either approach, Anthony would have to abandon his current working set to perform the transformation because existing tools only allow one working set at a time.

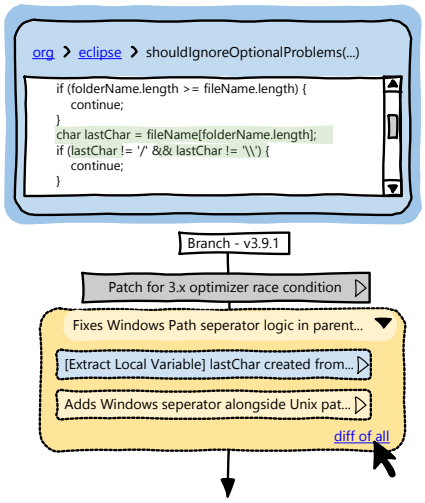

Fig. 4. An inline diff visualization of a squashed commit bubble. The diff is presented on an already-visible code bubble present within in the current task context to minimize context switching.

This interruption would have several consequences. Anthony might forget details of his original task, for example, that `shouldIgnoreOptionalProblems` is in a broken state. Even if he does remember, he is now forced to recreate his working set, which duplicates work he has already done [19].

With Commit Bubbles, Anthony obtains the same result without breaking his flow; after writing the bugfix, he simply drags his two commits to the 3.9.1 working set and verifies the diff looks correct (Figures 3 and 4, respectively). Commit Bubbles keeps track of the different contexts and utilizes AST- and heuristic-based algorithms (see Section IV) to fluidly perform these history changes. Incidentally, since Anthony did not have to context switch, he quickly resumes properly handling the return value of `isParentOf`.

III. Emerging Results

We have conducted an informal pilot study ($n = 5$) in which participants from our lab were asked to perform an activity similar to the one described in Section II using Git[3]. We found that although participants could verbally describe the history revision they needed, they were unable to actually accomplish the task. The participants had to frequently switch between different tools, and the revision commands needed to perform the action were unintuitive (e.g., *rebase*).

Moreover, we found that participants verbally described operations in terms of code fragments, e.g., "move this conditional statement," yet version control tools required them to think in terms of lines. In contrast, when developers were first given the desired commit history (e.g. systematic strategy) and then instructed to make the necessary code changes, they were able to both write the code and create the history. We used this preliminary feedback as an inspiration for our vision.

[3]http://www.git-scm.com/

IV. Challenges

We articulate technical and usability challenges that we face in realizing our vision, and while doing so, reference related work that makes progress towards these areas.

Technical challenges. Two technical challenges require techniques that can automatically generate commits in real time and more robustly rearrange commit history. Generating commits is straightforward when the developer explicitly uses a tool, but research has shown that many developers perform operations manually, even when an automatic tool is available [5]. Towards these efforts, Negara and colleagues have proposed AST-based (rather than text-based) approaches to capturing edit sequences at an appropriate level of abstraction [6]. Similarly, Kirinuki and colleagues use past commits to determine if a proposed commit may include a tangled change [8]. Buse and colleagues offer an automatic technique for synthesizing human-readable documentation for arbitrary program differences [9]. Finally, Hayashi and colleagues have proposed an algorithm for refactoring commit histories without impacting the resulting code [20]. We think the advancement of these technical obstacles is critical to implementing our vision.

Usability challenges. We posit that blending coding activities with commit activities solves several key obstacles to generating best-practice commit histories, but the blending process introduces its own challenges. First, it is possible that keeping coding and commit activities as separate activities has its own cognitive benefits, and that these benefits are lost when these activities are unified. It is also possible that developers do not find the unification of these activities to support their flow; instead, they may be perceived as interruptions [13]. Relatedly, we must also consider if as-needed version control approaches can be complemented with other task-driven approaches, such as Mylyn [21]. In addition, we have not yet addressed the problem of "bubble overload", in which multiple bubbles for different tasks present themselves to the developer simultaneously, resulting in task contexts that are unmanageable for the developer. A possible avenue for addressing these issues may be recommendation systems, which appropriately reveal and dismiss information or tools as needed to support a particular task [22].

V. Conclusion

A significant barrier to creating systematic commit histories is that developers do not always work systematically; instead, they frequently work in an as-needed way. Our emerging results revealed that translating as-needed activities to systematic commit histories as a separate maintenance activity is a time-consuming and difficult process for developers to perform successfully. We proposed an alternative interaction model in which expensive cognitive context switches are minimized, and one in which exceptional situations are treated as a routine part of the developer's workflow. Finally, we demonstrated the promise of this approach by applying these principles to version control commits using Commit Bubbles. Our work demonstrates the potential of tools that adapt to developers' familiar workflow, rather than constraining developers to a tool's computational model.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1217700. We thank Steven P. Reiss for introducing us to Code Bubbles.

References

[1] J. Loeliger and M. McCullough, *Version Control with Git*, 2nd ed. O'Reilly Media, 2012.

[2] M. Ernst. (2014, Jul.) Version control concepts and best practices. [Online]. Available: http://homes.cs.washington.edu/~mernst/advice/version-control.html

[3] D. E. Perry, "The Inscape environment," in *ICSE '89*, May 1989, pp. 2–11.

[4] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," *Journal of Systems and Software*, vol. 7, no. 4, pp. 341–355, 1987.

[5] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, Jan. 2012.

[6] S. Negara, M. Vakilian, N. Chen, R. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?" in *ECOOP '12*, 2012, vol. 7313, pp. 79–103.

[7] N. Kusunoki, K. Hotta, Y. Higo, and S. Kusumoto, "How much do code repositories include peripheral modifications?" in *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, vol. 2, Dec. 2013, pp. 19–24.

[8] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! Are you committing tangled changes?" in *ICPC '14*, Jun. 2014, pp. 262–265.

[9] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *ASE '10*, Sep. 2010, pp. 33–42.

[10] D. L. Parnas and P. C. Clements, "A rational design process: How and why to fake it," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 251–257, Feb. 1986.

[11] S. Robertson. (2012, Feb.) Commit often, perfect later, publish once: Git best practices. [Online]. Available: http://sethrobertson.github.io/GitBestPractices/

[12] S. Perez De Rosso and D. Jackson, "What's wrong with Git?" in *Onward! '13*, Oct. 2013, pp. 37–52.

[13] C. Parnin and S. Rugaber, "Programmer information needs after memory failure," in *ICPC '12*, Jun. 2012, pp. 123–132.

[14] M. Ackerman, "The intellectual challenge of CSCW: The gap between social requirements and technical feasibility," *Human-Computer Interaction*, vol. 15, no. 2, pp. 179–203, Sep. 2000.

[15] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, "Code Bubbles: A working set-based interface for code understanding and maintenance," in *CHI '10*, Apr. 2010, pp. 2503–2512.

[16] R. DeLine and K. Rowan, "Code Canvas: Zooming towards better development environments," in *ICSE '10*, vol. 2, May 2010, pp. 207–210.

[17] A. Z. Henley and S. D. Fleming, "The Patchworks code editor," in *CHI '14*, Apr. 2014, pp. 2511–2520.

[18] M. J. Coblenz, A. J. Ko, and B. A. Myers, "JASPER: An Eclipse plug-in to facilitate software maintenance tasks," in *OOPSLA Workshop: Eclipse '06*, Oct. 2006, pp. 65–69.

[19] A. Ko, B. Myers, M. Coblenz, and H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[20] S. Hayashi, T. Omori, T. Zenmyo, K. Maruyama, and M. Saeki, "Refactoring edit history of source code," in *ICSM '12*, Sep. 2012, pp. 617–620.

[21] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *FSE '06*, Nov. 2006, pp. 1–11.

[22] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić, "The emergent structure of development tasks," in *ECOOP '05*, Jul. 2005, pp. 33–48.