Commit Bubbles

Titus Barik, Kevin Lubick, Emerson Murphy-Hill
North Carolina State University, USA
{tbarik,kjlubick}@ncsu.edu, emerson@csc.ncsu.edu

***Abstract*—When working with version control systems, developers are expected to follow a set of best practices, such as constructing systematic commit histories. These systematic commit histories should show well-defined steps with logical forward progress. Yet, the process by which developers write source code is frequently evolutionary, or as-needed, rather than systematic. We argue that the as-needed strategy that developers frequently use to write code is incompatible with the systematic strategy that is necessary to satisfy version control best practices, resulting in non-ideal commit histories. Our vision is a development model that reconciles as-needed activities with systematic commit activities. Our proposed tool, Commit Bubbles, operationalizes this vision by extending the working set metaphor used by Code Bubbles to support commit activities within the same environment that developers use to edit code. 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.**

I. Motivation

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 [1]. 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 [2].

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 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, and that 40% of test fixes involve changes to the tests themselves [6]. 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 [7], and as incomplete or incorrect commit messages, which fail to capture the full description of the change [8], [5]. Both of these issues are obstacles to supporting downstream change management tasks, such as merging commits between branches, and conducting effective code reviews [7].

A potential solution is to untangle commit histories into systematic histories using history revision operations offered by modern version control systems[1]. For example, a developer can theoretically reorder or delete commits using *rebase*. However, in practice, this procedure is intricate and difficult to perform correctly [9].

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 [10]. 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 failure-prone, especially after interruption [11]. Second, history revision as supported in tools today takes the perspective that developers are able 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 [12].

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 [13]. We postulate that developers can benefit from reasoning about commit activities in the same way. Our proposed extension, *Commit Bubbles*, supports developers by a) blending coding and commit activities as a unified activity, 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." [9]

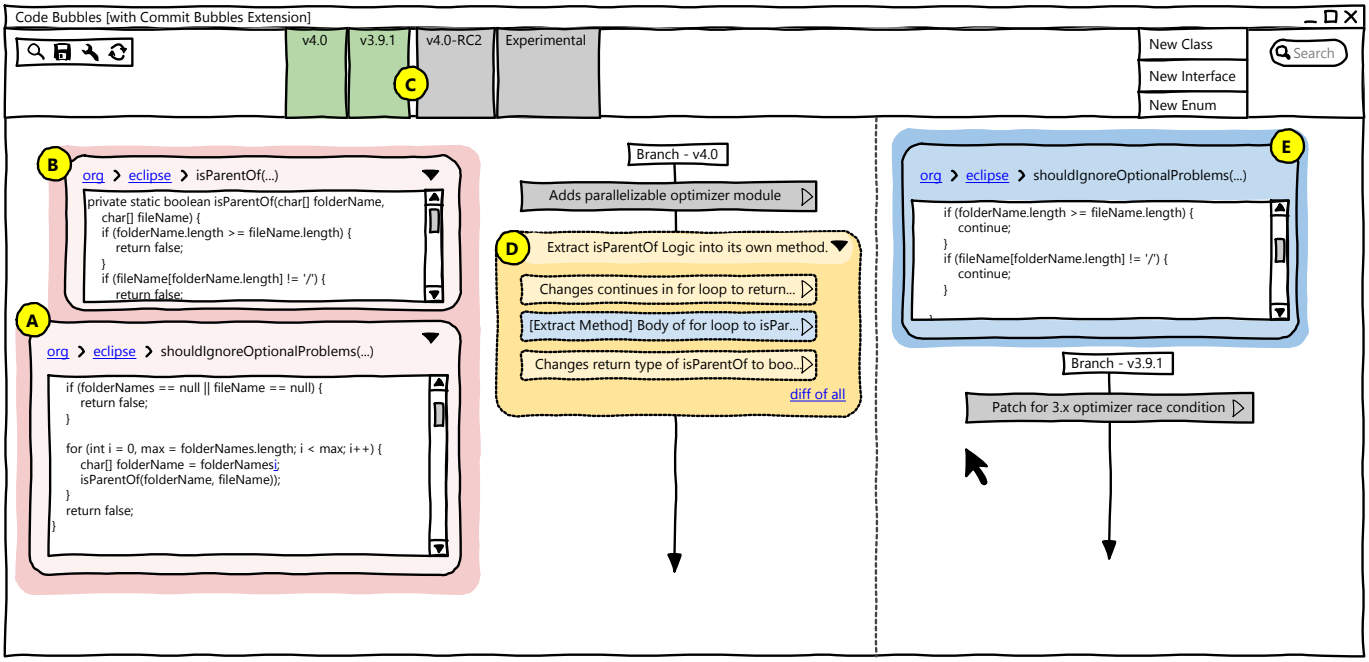

Fig. 1. A mockup of Code Bubbles, extended with Commit Bubble elements. (A), (B) and (E) are code bubbles, (C) is a *task context*, that is, a panel of working sets, (D) is an expanded commit bubble, consisting of three squashed commit bubbles

II. Commit Bubbles

The groundwork, Code Bubbles, through which we implement our idea, uses a metaphor of light-weight editable code fragments, which are displayed on a canvas in the editor as *bubbles* (Figure 1). These bubbles can be grouped together to form concurrently visible *working sets*, or sets of bubbles that relate to each other in a way that is useful to the developer. Our idea leverages the bubbles metaphor to aid developers in remembering all of the change activities related to a particular commit.

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. Promisingly, existing research has shown this metaphor can be extended to other domains. For example, in Debugger Canvas, DeLine and colleagues [14] augmented the bubbles metaphor to support run-time debugging activities. They incorporated visualizations for temporal aspects of debugging activities and automatic branches for multi-threaded programs. We assert one reason both of these activities (coding and debugging) are well-suited for this metaphor because the interaction aligns with the as-needed strategy that developers frequently use.

Given the success of Debugger Canvas, we postulate that developers can benefit from the bubbles metaphor when reasoning about version control commits. As with coding and debugging, commit activities also require reasoning about multiple working sets, such as when branching or when comparing change between two versions. And similar to debugging, the temporal aspect of commits is important in order to tell a logical, cohesive narrative.

A significant issue not addressed by existing bubbles metaphors is that commit activities require translation between strategies. With version control, developers need to eventually translate as-needed code changes into a history that is systematic, a process that is challenging for developers with existing tools [10], [9]. Rather than treating history revision as an independent activity, our novel idea is to support an as-needed developer workflow through a unified interaction mode that allows them to fluidly revise their history as they code.

In the next section, we narrate a user experience that demonstrates how a developer would combine Code Bubbles and Commit Bubbles in their daily programming activities to correct a defect present in multiple versions of the code base.

A. 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 within one of the loops, which he decides to refactor to `isParentOf` using a combination of the extract method feature of his editor and some manual cleanup (Figure 1B). Simultaneously, 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 combines them into one larger commit bubble He edits

[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

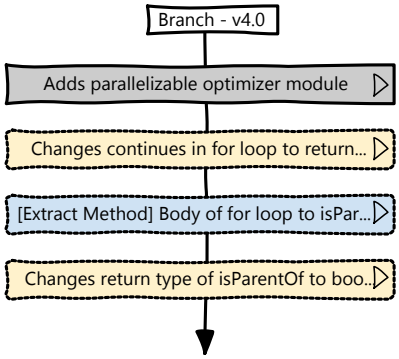

Fig. 2. Commit bubbles generated automatically while developing code. A solid outline indicates the commit bubble has been stored to public history, dashed outlines indicate commits that can be reordered or deleted

the autogenerated commit message to better reflect his intent, as seen in Figure 1D.

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 (`'/'`). Anthony adds a condition for the Windows line separator (`'\\'`) and creates a local variable to reduce redundant code. He squashes these two commit bubbles into one (Figure 3A). This bug exists in the 3.9.1 branch as well, which he verifies instantiating the corresponding code bubble (Figure 1E).

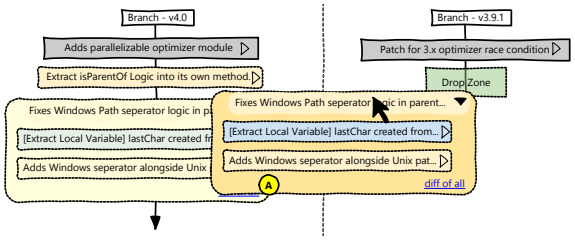

Fig. 3. Copying commit bubbles across working sets.

In a traditional version control system, 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 1D) 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 "save your all your edits, then re-introduce them in logical chunks, committing as you go" [1]. More sophisticated approaches exist, but they are even more burdensome and error-prone because it relies on complex and non-obvious git commands [9]. Worst of all, in either approach, he would have to abandon his current working set to perform the transformation because existing tools only allow one working set at a time.

This interruption would have several consequences. First,

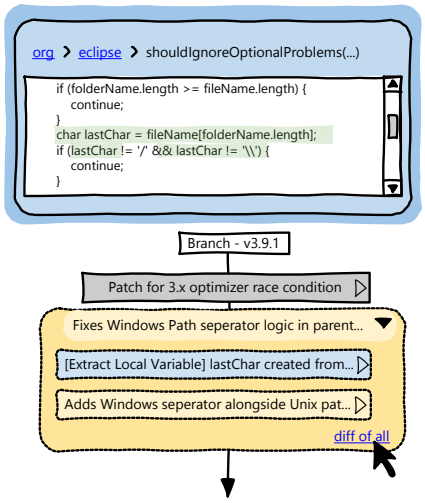

Fig. 4. An inline diff visualization of a squashed commit bubble. The diff is presented on an already-visible code bubble present within in the current task context.

Anthony might forget details of his original task, for example, that `shouldIgnoreOptionalProblems` is in a broken state. Second, even if he does remember, he now needs to recreate his working set which duplicates work he had already done.

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 a small, informal pilot study in which participants from our lab were asked to perform an activity similar to the one described in Section II-A using Git[3], where developers attempted to create a revised history after completing the task. From this pilot study, we found that even though participants could verbally describe the change needing to be performed, they were unable to actually accomplish the task because they frequently had to switch context between different tools; the operations required to perform the change were significantly misaligned than their verbal description of the task, and that there was no intermediate feedback by the tool when performing the transformations (such as through the Git *rebase* history revision command).

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)

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

and then instructed to make the necessary code changes, they were able to complete the task successfully. We used this preliminary feedback as an inspiration for our vision.

IV. Challenges

In this section, we articulate several 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 of note require techniques that can automatically generate commits in real time and more robustly rearange commit history. Generating commits is straight forward 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 uses past commits to determine if a proposed commit may include a tangled change [7]. Buse and colleagues offer an automatic technique for synthesizing human-readable documentation for arbitrary program differences [8]. Finally, we are particularly encouraged by the impressive work of Hayashi and colleagues, who have proposed an algorithm for refactoring commit histories without impacting the resulting code [15]. We think the advancement of these technical obstacles are 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 supports their flow; instead, they may be perceived as interruptions [11]. In addition, we have not yet addressed the the problem of "bubble" overload, in which multiple bubbles for different tasks present themselves to the developer simultaneously, resulting in task contexts unmanageable for the developer. A possible avenue for addressing these issues may be recommendation systems, which appropriately reveal and dismiss bubbles as needed to support a particular task [16], [17]. Importantly, we tacitly assumed that the future of IDEs will use the bubbles metaphor; further user studies are needed to validate the appropriateness of this metaphor as a general development environment for different tasks.

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 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 the Software Engineering group at ABB Corporate Research for their funding and support. We also thank Steven P. Reiss at Brown University for introducing us to Code Bubbles.

References

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

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

[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 2012*, J. Noble, Ed. Springer Berlin Heidelberg, 2012, vol. 7313, pp. 79–103.

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

[8] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *ASE '10*, Sep. 2010, p. 33.

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

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

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

[12] 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.

[13] 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, p. 2503.

[14] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, "Debugger canvas: Industrial experience with the code bubbles paradigm," in *ICSE '12*, 2012, pp. 1064–1073.

[15] 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.

[16] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation Systems for Software Engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, Jul. 2010.

[17] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić, "The emergent structure of development tasks," in *ECOOP '05*, A. P. Black, Ed., vol. 3586. Berlin, Heidelberg: Springer Berlin Heidelberg, Jul. 2005, pp. 33–48.