Glanceable Code History: Visualizing Student Code for Better Instructor Feedback

Caitlin Cassidy, Max Goldman, Robert C. Miller
MIT CSAIL
Cambridge, MA USA
{ccassidy,maxg,rcm}@mit.edu

ABSTRACT
Immediate, individualized feedback on their code helps students learning to program. However, even in short, focused exercises in active learning, teachers do not have much time to write feedback. In addition, only looking at a student's final code hides a lot of the students' learning and discovering process. We created a glanceable code history visualization that enables teachers to view a student's entire coding history quickly and efficiently. A preliminary user study shows that this visualization captures previously unseen information that allows teachers to give students better grades and give students longer feedback and better feedback that focuses not just on their final code, but all their code in between.

ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

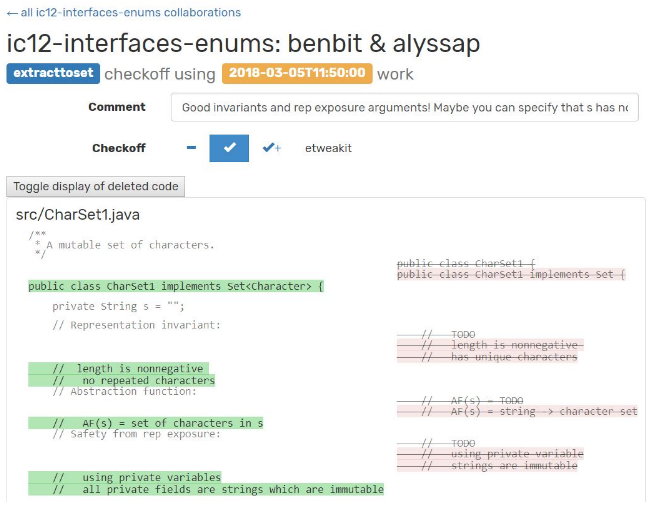
Author Keywords
learning at scale; computer programming; visualizations

INTRODUCTION
Research on learning has shown that immediate, individualized feedback helps student learn [1, 4, 10]. However, giving such feedback for students learning to program is difficult. Novice code is difficult for teachers to understand and each student has different misunderstandings about their code.

One-on-one tutoring from a programming expert is a powerful way to give a student individualized feedback. The expert can take time to fully understand the student's code and to listen to the student explain his/her thought processes. Then, the expert can address each of their misconceptions and give feedback about variable names, design patterns, etc. However, one-on-one tutoring is not feasible in large university classes, where there are far fewer teachers than students. With existing systems, teachers do not have time to take a fine-grained look at student code and give in-depth feedback on it.

L@S 2018, June 26–28, 2018, London, United Kingdom

ISBN 978-1-4503-5886-6/18/06. . . 15.00
DOI: https://doi.org/10.1145/3231644.3231680


← all ic12-interfaces-enums collaborations

ic12-interfaces-enums: benbit & alyssap

extracttoset checkoff using 2018-03-05T11:50:00 work

Comment Good invariants and rep exposure arguments! Maybe you can specify that s has nc

Checkoff – ✓ ✓+ etweakit

Toggle display of deleted code

src/CharSet1.java

```
/**
 * A mutable set of characters.
 */
                                                    ~~public class CharSet1 {~~
                                                    ~~public class CharSet1 implements Set {~~
public class CharSet1 implements Set<Character> {

    private String s = "";
    // Representation invariant:
                                                    ~~//  TODO~~
                                                    ~~//  length is nonnegative~~
                                                    ~~//  has unique characters~~

    //  length is nonnegative
    //   no repeated characters
    // Abstraction function:
                                                    ~~//  AF(s) = TODO~~
                                                    ~~//  AF(s) = string -> character set~~

    //   AF(s) = set of characters in s
    // Safety from rep exposure:
                                                    ~~//  TODO~~
                                                    ~~//  using private variable~~
                                                    ~~//  strings are immutable~~

    //   using private variables
    //   all private fields are strings which are immutable
```

Figure 1. Glanceable code history for an exercise asking students to change the CharSet1 class to implement the Set interface and write internal documentation for the CharSet1 datatype.

Computer science teachers have tried many ways to increase the amount of feedback given to students. Researchers have developed systems to automatically grade problem sets, give feedback about variables, and propagate teacher feedback to many students at once, all at scale [3, 6, 9]. Efforts also include clicker questions in lecture and a class Q&A site like Piazza, which improve feedback but have drawbacks, such as students not knowing how to phrase their questions or not knowing when to ask questions. Codeopticon, a one-to-many tutoring interface where teachers can view student code in real-time and proactively help students they see are struggling, can address these issues [8]. Many classrooms also use active learning, with some versions getting feedback to students quickly and while they are still learning the topic [11, 2]. In computer science classrooms, active learning may include short and focused coding exercises during class meetings, but teachers have limited time to write that feedback.

We propose a new approach, shown in Figure 1, called glanceable code history to improve feedback on programming during active learning (we focus on short coding exercises in particular). This approach is a visualization that displays not only what code students submitted at the end of the exercise, but also highlights steps that students took along the way. This allows teachers to understand how a student's final code evolved without watching the students the entire time or navigating a more complex visualization [13, 14]. Teachers can notice what misconceptions students first have then see whether the students later remedy them and teachers can learn more about that student's coding abilities so that they can give more in-depth and more personalized feedback.

We used glanceable code history in a preliminary study in an MIT active learning classroom and found that for almost all students, the visualization led to identical or better grading and feedback for students. In multiple instances, the students received a better grade and/or better feedback when teachers used glanceable code history.

IMPLEMENTATION

We implement glanceable code history as part of Constellation, an existing system for pair-programming on in-class exercises [7]. The system was designed for an MIT class – 6.031 – that teaches "fundamental principles and techniques of software development" [5, 12]. This class uses active learning in almost all class meetings: teachers give 10-15 minute coding exercises where students collaborate on Java code in pairs through Eclipse. Figure 2 illustrates Constellation's current visualization, which only shows final code: any student-added code is highlighted in yellow and provided code has gray text.

Our visualization replaces Constellation's current visualization and shows the critical parts of a student pair's code history during these active learning exercises concisely to ensure it's glanceable. The algorithm records individual edits, groups them to create snapshots of student work over time, captures differences between those snapshots, combines the differences together, and displays them. Here is a detailed breakdown:

Operations

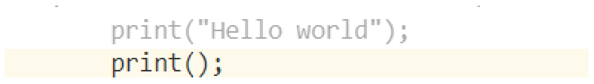
Constellation already includes an Eclipse plug-in for students that stores every operation performed by each student. These operations include the text that was typed or deleted and a timestamp, and form the basis for the algorithm.

Snapshots

From the operations, we generate a series of snapshots of the student's code at various times. We separate snapshots when the time between a previous operation and the next operation is greater than a given threshold. The threshold ensures that we don't include spurious information, like typos. After calibrating with many different student pairs and exercises, we chose a default threshold of 10 seconds (10000 milliseconds).

Diffs

Next we use a standard text diff, where the differences between two pieces of texts are represented as blocks of insertions, deletions, and unchanged text. A standard text diff uses a line-based diff, so that any change in a line is represented as the deletion of the entire line and an insertion of the new, edited line, and we use a line-based diff as well because we found that it was easiest for teachers to understand quickly. We generate a standard text diff between every two consecutive snapshots – for example, given ten snapshots, we have nine diffs.



```
print("Hello world");
print();
```

Figure 2. Constellation's current visualization only shows final code.

Baseline

Typically, teachers provide students with code before the exercise and it's important to distinguish provided code from all other code. To do this, we add a diff at the very beginning that is the provided code represented as one unchanged block to be our baseline. We also mark it with a special marker, "provided", and carry that marker throughout the next step so that we can display provided code differently.

Combined Diff

We then combine all of our individual diffs together to create a single diff that still has blocks of insertions, deletions, and unchanged text. To do this, we iterate over each individual diff in turn and combine it with the previous diffs. For any added parts, we insert the block and mark it as added. For any removed parts, we scan through all blocks that contain the removed text and mark those block(s) as removed. Since we keep all removed code from past diffs, our combined diff will contain characters that future diffs do not know about. The algorithm's main challenge is skipping over those parts when combining those future diffs. Figure 3 shows how the algorithm transforms operations into a combined diff.

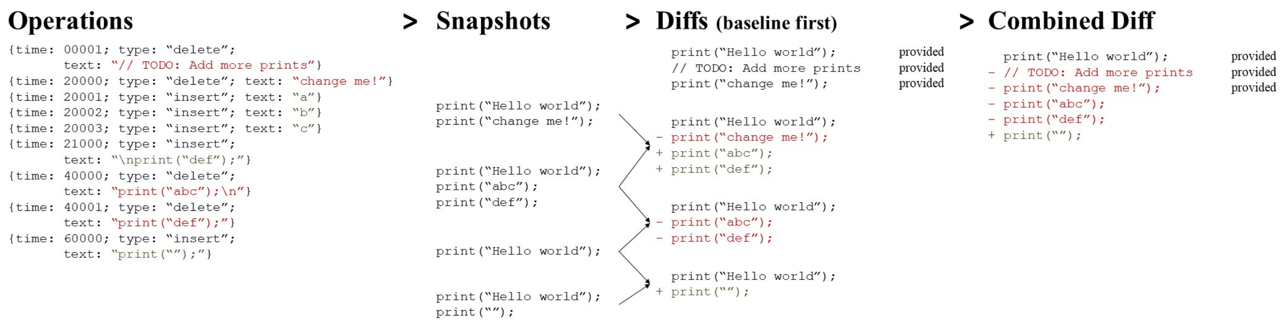
Display

We then display the combined diff for teachers. We found that teachers often want to pay attention to the pair's final code first then deleted code afterwards (since the final code gives them context for the code's history), so we show all final code (both provided code and student code) on the left and deleted code (both provided code that was deleted and code that students added then deleted later) on the right. We also allow them to only see final code with a toggle button, shown in Figure 1.

Figure 4 looks at an exercise where a student adds two new print statements and then deletes them, highlighting how glanceable code history gives the teacher a far better understanding of a students' work. Figure 5 gives a key explaining the display. Any code that students added and still have at the end of the exercise is highlighted in green. Provided code is gray text, and is struck through if the student deleted it. Code that students added then deleted at some point later has gray text, is struck through, and is highlighted red to distinguish it from deleted provided code.

PRELIMINARY USER STUDY

We conducted a preliminary user study to see whether the glanceable code history visualization improved the feedback received by students by studying one exercise in the MIT 6.031 classroom. Student pairs were given buggy Sudoku puzzle solver written in Java, and a failing test case: trying to solve the empty puzzle would fail. Students were asked to reduce this test case, so that instead of trying to solve the empty puzzle (9x9=81 blanks), the solver was shown to fail on a puzzle with



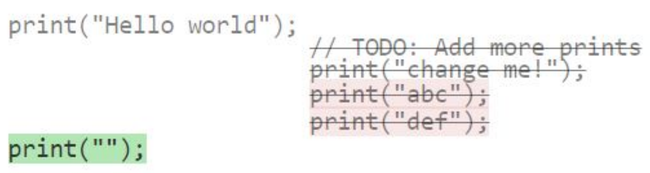
```
{time: 00001; type: "delete";
       text: "// TODO: Add more prints"}
{time: 20000; type: "delete"; text: "change me!"}
{time: 20001; type: "insert"; text: "a"}
{time: 20002; type: "insert"; text: "b"}
{time: 20003; type: "insert"; text: "c"}
{time: 21000; type: "insert";
       text: "\nprint("def");"}
{time: 40000; type: "delete";
       text: "print("abc");\n"}
{time: 40001; type: "delete";
       text: "print("def");"}
{time: 60000; type: "insert";
       text: "print("");"}
```

```
print("Hello world");
print("change me!");

print("Hello world");
print("abc");
print("def");

print("Hello world");

print("Hello world");
print("");
```

```
  print("Hello world");          provided
  // TODO: Add more prints       provided
  print("change me!");           provided

  print("Hello world");
- print("change me!");
+ print("abc");
+ print("def");

  print("Hello world");
- print("abc");
- print("def");

  print("Hello world");
+ print("");
```

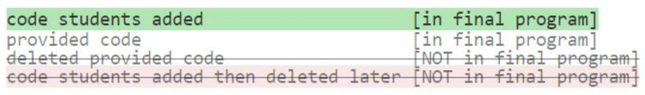
```
  print("Hello world");          provided
- // TODO: Add more prints       provided
- print("change me!");           provided
- print("abc");
- print("def");
+ print("");
```

Figure 3. Individual insert and delete operations transform into snapshots, then diffs, then a single combined diff with appropriate labels.



```
print("Hello world");
                          ~~// TODO: Add more prints~~
                          ~~print("change me!");~~
                          ~~print("abc");~~
                          ~~print("def");~~
print("");
```

Figure 4. Glanceable code history reveals previously unseen student work, allowing teachers to better understand how students program.



code students added	[in final program]
provided code	[in final program]
~~deleted provided code~~	~~[NOT in final program]~~
~~code students added then deleted later~~	~~[NOT in final program]~~

Figure 5. A key explaining the display of glanceable code history.

fewer blank squares. Our users were the Teaching Assistants (TAs) for 6.031, who grade active learning exercises multiple times in each class (each TA grading 12-18 pairs per exercise. TAs give each pair a grade of a -, (no or little progress and no credit), ✓ (sufficient progress), or ✓+ (excellent progress), and almost always write a comment as feedback.

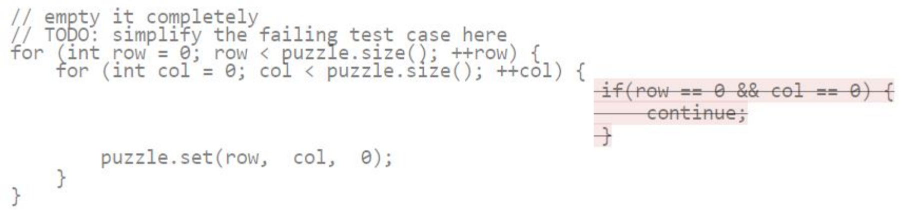
For our study, the TAs followed the same process as normal class meetings, except that they graded the first half of their students using the glanceable code history visualization and the second half using Constellation's current visualization. Both visualizations were integrated with the normal interface shown in Figure 1. TAs have a lot of experience with the current visualization, so we could not control for the amount of exposure; rather, we wanted them to use each visualization on an equal number of pairs. We also randomly selected some students to be graded twice, once with the current visualization and once with our glanceable code history. There were 109 student pairs and 144 grades given out (8 TAs with 18 students per TA), with 35 pairs graded twice (32% of the total pairs).

Based on a qualitative review on the collected grades and comments, we found that the visualization led to identical or better grading and feedback for students. As evidence of the visualization allowing TAs to learn more about student's entire coding process, we saw multiple examples where glanceable code history captured important information that the current interface did not. Figure 6 shows the code of a pair who received a - score and the feedback "Test case not simplified" when being graded without code history, since the current visualization would only show the information on the left side. Yet that pair received a ✓ score and the feedback "Good start. You were on the right track with filling in less 0's" by the TA who used glanceable code history. That TA saw evidence of student progress and had more code on which to give feedback. Figure 7 shows student code that illustrates how the current visualization would show only one attempt at solving the exercise while the additional code history highlights the pair's exploration of different test case reductions and explains how they arrived at their final solution.

We also saw improved feedback given by TAs who used the visualization. One pair received the feedback "Excellent strategy and regression tests!!" by the TA who used our visualization, versus "Good job" by the other. While this could be due to individual differences between how TAs give feedback, we saw that on average, TAs used 49.4 characters in their feedback when they were using glanceable code history, and 43.2 characters without. 82% of feedback given using the current visual had more than 10 characters, while 90% of feedback given using glanceable code history were at least that long.

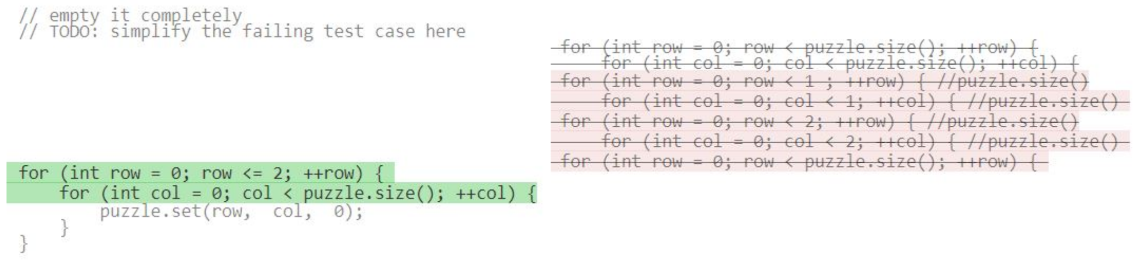
We found two cases where a pair received a worse grade when the TA used glanceable code history versus a TA using the current visualization. For one pair, their history had many different additions and deletions spread throughout the file and that may have made their code more difficult to read. To address this, our next iterations will explore altering the algorithm to prune the code history if it gets too large and making it easier for TAs to remove the display of history. For the second pair, we hypothesized that they received a lower score because the current visualization has syntax highlighting, which made a code comment very easy to see, whereas code history does not incorporate any syntax highlighting. We will try incorporating syntax highlighting in the next iteration to improve code readability.

When TAs evaluated the glanceable code history interface, they found it to be generally useful. While they all graded some pairs of students without needing to look at deleted code, they found certain pairs where it became very helpful. One TA wrote "I was able to give credit to a pair who deleted some good progress with the red/green visualization. Seeing some of the deleted code helped with figuring out the pair's train of



```
// empty it completely
// TODO: simplify the failing test case here
for (int row = 0; row < puzzle.size(); ++row) {
    for (int col = 0; col < puzzle.size(); ++col) {
                                                   ~~if(row == 0 && col == 0) {~~
                                                   ~~    continue;~~
                                                   ~~}~~
        puzzle.set(row,  col,  0);
    }
}
```

Figure 6. The glanceable code history visualization raised this pair's score from a - to a ✓ since the history highlights some student progress.



```
// empty it completely
// TODO: simplify the failing test case here
                                                   ~~for (int row = 0; row < puzzle.size(); ++row) {~~
                                                   ~~    for (int col = 0; col < puzzle.size(); ++col) {~~
                                                   ~~for (int row = 0; row < 1 ; ++row) { //puzzle.size()~~
                                                   ~~    for (int col = 0; col < 1; ++col) { //puzzle.size()~~
                                                   ~~for (int row = 0; row < 2; ++row) { //puzzle.size()~~
                                                   ~~    for (int col = 0; col < 2; ++col) { //puzzle.size()~~
                                                   ~~for (int row = 0; row < puzzle.size(); ++row) {~~
for (int row = 0; row <= 2; ++row) {
    for (int col = 0; col < puzzle.size(); ++col) {
        puzzle.set(row,  col,  0);
    }
}
```

Figure 7. Glanceable code history highlights this pair's exploration of many different solutions.

thought" and another wrote "Cool to see do-then-undo actions. Had clear case where final code doesn't tell the whole story (ended with starting code, but tried something in between)".

CONCLUSION

Glanceable code history visualizes student code and is tailored specifically to improve feedback given by teachers after short, active learning exercises in programming. A preliminary user study has shown that this visualization allows teachers to see and understand the steps a student took before arriving at their final code, and helps teachers give more in-depth and stronger feedback. Future work will include prototyping other visualizations of code history and iterating on glanceable code history based on user feedback. Glanceable code history can allow teachers to better understand a student's entire coding process and allow students to receive more helpful feedback than is currently possible.

REFERENCES

1. S. A. Ambrose, M. W. Bridges, M. DiPietro, M. C. Lovett, and M. K. Norman. 2010. *How Learning Works: Seven Research-Based Principles for Smart Teaching (1 ed.)*. Jossey-Bass.
2. C. Bonwell and J. A. Eison. 1991. *Active Learning: Creating Excitement in the Classroom*. ERIC Clearinghouse on Higher Education.
3. B. Cheang, A. Kurnia, A. Lim, and W. Oon. 2003. On automated grading of programming assignments in an academic institution. *Computers Education* 41, 2 (2003), 121–131.
4. National Research Council. 2000. *How people learn: Brain, mind, experience, and school: Expanded edition.* National Academies Press.
5. MIT EECS. 2018. 6.031 Software Construction Class Website. (2018). `http://web.mit.edu/6.031/www/sp18/`.
6. E. Glassman, L. Fischer, J. Scott, and R. C. Miller. 2015. Foobaz: Variable name feedback for student code at scale. In *UIST 2015 (ACM)*. 609–617.
7. M. Goldman. 2018. Constellation. (2018). `http://maxg.github.io/constellation`.
8. P. J. Guo. 2015. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *UIST 2015 (ACM)*. 599–608.
9. A. Head, E. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Learning@ Scale 2017 (ACM)*. 89–98.
10. K. Koile and D. Singer. 2006. Improving learning in CS1 via tablet-PC-based in-class assessment. In *Proceedings of the second international workshop on Computing education researchy (ACM)*. 119–126.
11. C. Meyers and T. B. Jones. 1993. *Promoting Active Learning. Strategies for the College Classroom.* Jossey-Bass Inc.
12. MIT. 2018. 6.031 Course Description. (2018). `http://student.mit.edu/catalog/search.cgi?search=6.031&style=verbatim`.
13. J. Park, Y. H. Park, S. Kim, and A. Oh. 2017. Eliph: Effective Visualization of Code History for Peer Assessment in Programming Education. In *CSCW 2017 (ACM)*. 458–467.
14. Y. Yoon, B. A. Myers, and S. Koo. 2013. Visualization of fine-grained code change history. In *VLHCC 2013 (IEEE)*. 119–126.