

# How Live Coding Affects Developers' Coding Behavior

Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, Jan Borchers  
RWTH Aachen University  
52062 Aachen, Germany  
{kraemer, kurz, karrer, borchers}@cs.rwth-aachen.de

**Abstract**—We report on the behavior of developers working with a live coding environment, which provides information about a program's execution immediately after each change to the source code. The live coding environment we used shows information about each individual source code line, e.g., changed variable values or truth values of conditions. In comparison to developers working in a non-live environment, those working live found and fixed bugs they introduced significantly faster. Further, working live encouraged developers to switch between editing and debugging phases more frequently.

## I. INTRODUCTION

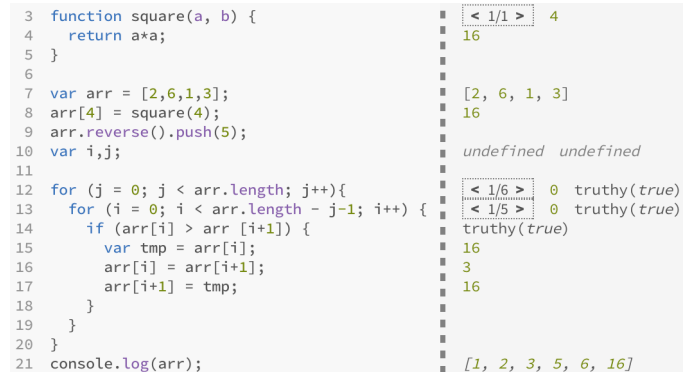
Tanimoto [1] coined the term *liveness* to classify programming systems by the immediacy of the feedback they provide. Modern IDEs, such as Eclipse or Xcode, already provide feedback about syntax and compilation errors immediately after a change to the source code. This feature is called continuous compilation. In contrast, we only call a programming environment *live*, if it also informs the developer about program execution and updates this information immediately after the source code is changed. Such information includes, e.g., the values of variables after assignments, the values of conditions, the parameters passed to methods, or the chronological order of methods called.

Because in a live coding environment information about program execution is provided immediately after each change, developers should have less need to switch to a dedicated debugging phase to inspect the execution of a program. Instead, they can spot bugs immediately after they are introduced. Not only could this make developers faster, it could also help them comprehend their or others' source code in more depth.

In this paper, we make the following contributions: We compare the behavior of JavaScript developers working with a live coding environment to the behavior of developers working without it. This comparison is evaluated quantitatively and qualitatively to find support for the potential benefits of live coding outlined above. In particular, we found participants in the live coding condition to notice and fix bugs they introduced faster. Further, in the live coding condition, source code edits were more evenly spread over the time working on a task.

## II. RELATED WORK

Tanimoto [1] differentiated four levels of liveness in programming systems: (1) Non-executable program descriptions; (2) executable program descriptions, e.g., most source code in current development environments; (3) program descriptions that run automatically whenever they are changed; and (4)



```
3 function square(a, b) {
4   return a*a;
5 }
6
7 var arr = [2,6,1,3];
8 arr[4] = square(4);
9 arr.reverse().push(5);
10 var i,j;
11
12 for (j = 0; j < arr.length; j++){
13   for (i = 0; i < arr.length - j-1; i++) {
14     if (arr[i] > arr [i+1]) {
15       var tmp = arr[i];
16       arr[i] = arr[i+1];
17       arr[i+1] = tmp;
18     }
19   }
20 }
21 console.log(arr);
```

Execution console output:

```
< 1/1 > 4
16
[2, 6, 1, 3]
16
undefined undefined
< 1/6 > 0 truthy(true)
< 1/5 > 0 truthy(true)
truthy(true)
16
3
16
[1, 2, 3, 5, 6, 16]
```

Fig. 1. Our experimental prototype displays information about the program execution on the right side of a source code editor, and updates it after every keypress. The prototype shows variable values after each assignment, truth values of conditions, log messages and function parameters, and it supports inspecting multiple loop iterations and function calls.

systems that keep the program running so that changes to the source code affect its behavior in response to future events. Recently, Tanimoto [2] added two levels: programming systems that try to predict the developer's intention by (5) evaluating nearby versions, or (6) guessing higher level goals and large chunks of source code. In this paper, when we refer to live coding, we mean systems of at least level 3.

Live systems date back to Visicalc, the first spreadsheet environment, which already achieved level 3 liveness. Later, Burnett et al. [3] discussed how spreadsheet-like environments, in particular Forms/3, can be enhanced to also achieve level 4 liveness. Smith and Ungar introduced the Self programming language [4] that comes with a graphical user interface in which developers can manipulate all objects directly, and that thus achieves level 4 liveness. Edwards [5] showed that live coding environments can also be built for Java, a traditional object-oriented programming language.

Evaluations of *how* live coding changes developers' behavior are less common. Wilcox et al. [6] compared a live and a non-live version of Forms/3 in terms of the time developers needed to find seeded bugs in an application. They found that the effectiveness of live coding depends on the task and the kind of errors to fix, as some bugs were fixed significantly faster in one condition, some in the other, and for other bugs no difference could be found. Our evaluation complements these results in two ways: Firstly, we analyze code creation tasks instead of seeding bugs into existing source code, and secondly, we analyze a textual programming language.

Saff and Ernst [7] proposed *continuous testing*, in which test cases are run automatically after each change, showing the developer immediately which tests currently fail. Continuous testing can be considered a special case of live coding, providing one particular kind of information about the program’s execution. They compared the performance of developers using either an unmodified editor, only continuous compilation, or continuous compilation and continuous testing. Compared to the baseline, time constrained implementation tasks with a given test suite could be solved successfully by three times as many participants in the continuous testing condition and by twice as many in the continuous compilation only condition.

Choi et al. [8] presented Rehearse, a tool that allows developers to defer writing the actual body of a JavaScript function until it is first called. A developer can then implement it in an editor that executes every line after it was typed. This editor implements live coding, although a source code line can not be changed once it was executed without undoing all changes up to the implementation of that line.

Snell [9] presented a similar tool that executed every line immediately and showed how the variables in scope changed. Informal testing suggested that errors propagated less often and users felt less stressed when using the tool. We complement these results with a more thorough, formal evaluation.

Recently Bret Victor promoted the idea of live coding in a well-received talk<sup>1</sup>. His design has inspired several publicly available live coding systems, such as Light Table<sup>2</sup>, or one of the browser-based learning tools by the Kahn Academy<sup>3</sup>. We are not aware of a scientific evaluation of any of these systems.

### III. PROTOTYPE

To analyze how developers’ behavior changes when working in a live coding environment, we first implemented a live coding environment. Because participants in our study should have a realistic feeling of working live, the prototype had to be capable of providing execution information for JavaScript source code at interactive rates. To meet this requirement, a mock-up was unfeasible and we implemented a fully working live coding environment. We publish the prototype as an open-source project<sup>4</sup> to facilitate reuse in other studies.

The prototype consists of a Node.js<sup>5</sup> backend server that runs and traces the execution of JavaScript code, and a frontend that visualizes the runtime information gathered by the backend. The backend exposes its functionality via a WebSocket API. The frontend sends arbitrary JavaScript code to the backend for execution. The backend instruments this code and then executes it in a sandbox that is a separate node process. During instrumentation, callbacks are added to the code that send the relevant information from the sandbox to the backend, where they are chunked to reduce communication overhead, and forwarded to the frontend. This way, the backend remains responsive even if the executed JavaScript code is erroneous, and it can already send information back to the frontend while the process is still running. If new code arrives

```

! 14      tmp = arr[i];
      'tmp' was used before it was defined.
15      arr[i] = arr[i+1];
  
```

Fig. 2. Our implementation of continuous compilation shows error indicators next to the line numbers and error position markers in the line. Clicking the error indicator reveals a textual error description inline.

from the frontend while an old version of the code is still running, sandboxing also allows the backend to terminate old versions. Full documentation about information collected by the backend is included in the open-source release.

Because the source code is sent from the frontend to the backend as a string, the backend can currently not provide live coding for JavaScript projects that are spread across multiple files. The application is restarted after each change, so our prototype is not useful for applications that, e.g., require user input to reach the edited method. Also, sandboxing in our implementation does not provide a secure execution context, i.e., operations like file or database access will be executed in the instrumented version and will affect the actual system the backend is running on. The instrumentation additionally causes some performance degradation. However, as we designed both the tool and the study, we made sure these limitations would have no impact on the tasks in our study. In our trials the time between entering a character and seeing the updated execution information on screen was well below one second.

We implemented the frontend as an extension for Brackets<sup>6</sup>, an open-source JavaScript IDE. Our extension shows one line of information obtained from the live coding backend next to each line in Brackets’s source code editor (Fig. 1). If lines are executed more than once because they are inside a loop or function, the displayed execution can be selected using an iteration selector next to the loop or function declaration. This design is close to some of Brett Victor’s popular but so far unevaluated ideas on how to implement live coding for applications with no graphical output.

### IV. STUDY SETUP

We compared developers’ coding behavior between two conditions: in the experimental condition developers worked with our live coding prototype; in the control condition developers worked with the unmodified Brackets IDE. We assumed that developers using live coding could fix many bugs right away without a mental context switch. From this assumption we derived the following three hypotheses: (1) The average *total fix time*, the time between a developer introducing a bug and correcting it, is lower in the experimental condition. As a result of (1), (2) the task completion time for a given task is reduced in the experimental condition. Finally, we assumed that the possibilities offered by live coding would also encourage new coding strategies: (3) more participants in the experimental condition will adopt a coding strategy that involves finding and removing errors throughout the development process instead of mainly at its end.

Each participant had to solve three tasks. In each task, specific functionality had to be implemented in a Node.js

<sup>1</sup><http://worrydream.com/#!/InventingOnPrinciple>

<sup>2</sup><http://www.lighttable.com/>

<sup>3</sup><https://www.khanacademy.org/cs/programming>

<sup>4</sup><http://hci.rwth-aachen.de/livecoding>

<sup>5</sup><http://nodejs.org>

<sup>6</sup><http://brackets.io>, Sprint 24

command line application. In the first task, participants had to parse the RSS feed of a news website<sup>7</sup> using `sax-js`<sup>8</sup>, an open-source, stream-based XML parser. The difficulty in the first task was to understand the structure of the RSS feed and to use the parser without having experience with its API. In the second task, participants needed to convert between two different object representations of a date. This task provoked ‘programming slips’, i.e., errors caused by mixing up slightly different object representations (e.g., a property called `hours` instead of `hour`), or missing easy conversions (e.g., from local time to UTC). In the last task, we asked participants to implement Dijkstra’s algorithm. This task was focused on algorithmic correctness. A precise verbal explanation of the algorithm was included in the task description, copied from the corresponding Wikipedia article.

For each task, participants received both printed and PDF descriptions containing an explanation of the task, documentation about all data structures used, links to relevant API documentation, and short pieces of sample code. We also provided a code skeleton for each task including a test call to the functionality that had to be implemented.

We employed a between-groups study design. Node Inspector,<sup>9</sup> a graphical debugger interface for Node.js, was available to participants in both conditions; we provided a script to start the debugger without having to use the command line. To study the effects of live coding separately from those of continuous compilation, we provided continuous compilation for both conditions via a Brackets extension<sup>10</sup> (Fig. 2). Because we wanted to keep the scenario realistic, participants were allowed to use any external resources they wished, including web searches, code snippets from the web, or third-party JavaScript libraries (except those replacing functionality provided in `sax-js` for the first task).

## V. STUDY RESULTS

We analyzed data from 10 participants (9 male). All participants studied or had studied computer science, were 28.8 years old on average ( $SD = 8.95$ ), and reported at least 4.5 years of programming experience ( $M = 13.4, SD = 9.23$ ). The two groups were balanced in terms of programming experience (in years) and coding done per week.

To check our first hypothesis, we measured the total fix time for each bug. Therefore, one researcher annotated the introduction and fixing of each bug in all video recordings of the trials. A bug was recorded whenever newly added code led to erroneous behavior in the context of the current version of the source code. A problem with this method is that during a planned modification to the source code it will likely be in an erroneous state for a short time. In these situations, the developer usually is aware of those errors. To filter these planned, intermediate stages of code modifications, we removed bugs from the analysis that were fixed within 30 seconds after their introduction. A bug was never filtered, however, if the developer wrote unrelated code between the introduction of a bug and its correction. Of course, this filter

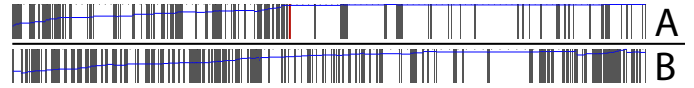


Fig. 3. The graphs show the behavior of developers A and B working on the third task. Time is plotted on the x-axis and file length (blue line) on the y-axis. Each individual change is indicated by a vertical gray bar. Developer A adopted the sequential strategy, with a visually discriminable debugging phase, developer B adopted the interleaved strategy.

is only a heuristic and will also filter out true bugs that are fixed within 30 seconds. Removing these true bugs increases the average total fix time of a trial. True bugs that are fixed within 30 seconds are more likely in the experimental condition, because participants received live execution information allowing them to react faster, so the filtering mechanism effectively benefits the control condition. In future studies, bug annotation accuracy could be further improved by using inter-rater reliability. We ended up with 205 annotated bugs.

An ANOVA (modeling task as within-groups factor and condition as between-groups factor) revealed no significant difference between conditions in terms of the number of bugs introduced (sum over all tasks, experimental:  $M = 19.1, SD = 4.8$ , control:  $M = 24.2, SD = 7.8, F(1, 19.2) = 1.36, p = 0.277$ ). We analyzed total bug fix times of all annotated bugs using a mixed-factor linear model in which we included condition, task, and the interaction of task and condition as fixed factors, and the participant nested within the condition as a random factor. The analysis reveals a significant decrease of the average total fix time in the live coding condition (in minutes, control:  $M = 18.5, SD = 21.0$ , experimental:  $M = 5.6, SD = 5.85, F(1, 9.124) = 6.94, p = 0.027$ ), no effect of task, and no interaction effect. This effect confirms our first hypothesis: developers find and fix bugs they introduced faster when using live coding.

In terms of task completion time, an ANOVA only showed a significant effect of task, not of condition ( $F(1, 8) = 2.794, p = 0.133$ ), and no interaction. Hence, we cannot confirm our second hypothesis. In contrast to the previous analysis, where the unit of analysis were the annotated bugs, in this analysis the unit of analysis were the participants. Consequently, the sample size for this test was much smaller, and an effect of condition might either be absent or too small to be detected using a parametric test. However, another possible explanation is that the effect of live coding on the task completion time is overshadowed by inter-subject differences. Our data can support this explanation: We observe that for each task the mean task completion time is lower in the live coding condition, but the standard deviation is the same order of magnitude as the means (sum over all tasks, in minutes, experimental:  $M = 135, SD = 44$  control:  $M = 204, SD = 92$ ).

To analyze our third hypothesis, we used a Kolmogorov-Smirnov test to compare the temporal distribution of edits in both conditions. We found a significant difference between these distributions ( $D = 0.05, p < 0.001$ ). This indicates that the strategies of participants are different between conditions. Qualitatively, we can identify two strategies for fixing bugs: Participants using the *sequential strategy* wrote down the source code completely and then tried to fix all bugs in one debugging session afterwards. The *interleaved strategy* was to test the code after each incremental change to keep it

<sup>7</sup><http://daringfireball.net>

<sup>8</sup><https://github.com/isaacs/sax-js>

<sup>9</sup><https://github.com/node-inspector/node-inspector>

<sup>10</sup><https://github.com/JoachimK/brackets-continuous-compilation>

free of bugs throughout the process. Figure 3 visualizes both strategies by plotting all changes of two participants in the third task over time. The editing and the debugging phase, in which only short bursts of edits are done to fix bugs, can be visually discriminated for developer A. In contrast, for developer B, who adopted the interleaved strategy, changes are more evenly spread over time, because editing and bug fixing phases were interleaved. We observed different use of these strategies depending on both the task and the condition. For the first task, all participants adopted the interleaved strategy. For the second task, in the control condition all participants but one used the sequential strategy; in the live coding condition, however, all participants but one used the interleaved strategy. The observations in the third task are equivalent to those in the second task, except that two participants from the control condition adopted the interleaved strategy. The K-S test and these observations confirm our third hypothesis.

The fact that developers in the control condition also chose to adapt the interleaved strategy for some tasks indicates that they felt a need to check their code’s correctness regularly. For the first task, we speculate that it was particularly easy to split up into sub-tasks, each of which could be individually tested. Backing up this speculation with experimental evidence is part of our future work. For the second task, which was designed to provoke programming slips, we can quantitatively confirm a positive effect of both live coding and the adoption of the interleaved strategy: An ANOVA modeling the condition and the strategy used as factors shows a significant reduction of task completion time in the live coding condition ( $F(1,6) = 13.19, p = 0.01$ ) and for the interleaved strategy ( $F(1,6) = 7.41, p = 0.03$ ). No significant effects were found for the third task. This indicates that only adopting the interleaved strategy, which live coding promotes, can already be beneficial for certain tasks.

To confirm that no major usability flaws of the prototype confounded our results, we asked participants in the live coding condition to fill out a *post-session questionnaire* including a System Usability Scale after the study. Our prototype achieved an average score of 81.4 ( $SD = 10.7$ ) on this scale, which can be considered excellent according to Bangor et al. [10].

In the post-session questionnaire, all participants but one agreed or strongly agreed that live coding has benefits for code understanding. Also, everyone felt that using the live coding environment had provided them with more confidence that their source code is correct. In individual questionnaires after each task, participants in the live coding condition agreed that live coding helped them solve the task.

## VI. PROTOTYPE LIMITATIONS

In the experimental condition all participants but one regularly added extra statements just to show the current value of a variable in the live coding pane. This indicates that our live coding frontend still did not provide all necessary information directly. The most frequent cause for these extra statements was object manipulation using one of the object’s methods. For such statements, the live coding view would show no information, since multiple concatenated method calls can appear in one line and it is unclear which object or return value should be visualized. For example, for line 9 in Figure 1 we

could show the value of `arr`, the result of `arr.reverse()`, or the return value of the full line. For all of the observed cases, participants were interested in the value of the object on which the last method was called, after the line was executed.

Problems of our current visualization were also noticeable in the questionnaire results: One participant found the continuous update of information in the live coding environment distracting, when the information was not currently relevant. Two participants indicated that the visualization of complex data, e.g., large chunks of the RSS feed in task 1, was complicated to parse quickly.

## VII. SUMMARY AND FUTURE WORK

We presented a study in which we showed that working in a live coding environment significantly decreases the average total fix time of bugs introduced while creating software. Additionally, live coding promotes adopting the interleaved coding strategy, i.e., regularly checking the code for correctness, which is especially effective for tasks that are prone to programming slips. We observed no decrease in task completion time when working live. To run this study we implemented a working live coding prototype that allows exploration of other visualizations for live coding systems.

We will explore new visualizations that allow incremental exploration of a larger amount of data, since our live coding prototype still does not present all information required. Furthermore, we will explore how live coding tools can visualize the call hierarchy to make the structure of nested method calls or callbacks more clear. Finally, we plan to evaluate the effect of combining live coding with continuous testing.

## VIII. ACKNOWLEDGMENTS

This work was funded in part by the German B-IT Foundation and by the German Government through its UMIC Excellence Cluster for Ultra-High Speed Mobile Information and Communication at RWTH Aachen University.

## REFERENCES

- [1] S. L. Tanimoto, “VIVA: A visual language for image processing,” *Journal of Visual Languages and Computing*, vol. 1, no. 2, 1990.
- [2] S. Tanimoto, “A perspective on the evolution of live programming,” in *1st Int. Workshop on Live Programming (LIVE)*, 2013.
- [3] M. M. Burnett, J. W. Atwood, Jr, and Z. T. Welch, “Implementing level 4 liveness in declarative visual programming languages,” in *Proc. VL ’98*. IEEE, 1998.
- [4] R. B. Smith and D. Ungar, “Programming as an Experience: The Inspiration for Self,” in *Proc. ECOOP ’95*, Aug. 1995.
- [5] J. Edwards, “Example centric programming,” *SIGPLAN Notices*, vol. 39, no. 12, 2004.
- [6] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook, “Does continuous visual feedback aid debugging in direct-manipulation programming systems?” in *Proc. CHI ’97*. ACM, 1997.
- [7] D. Saff and M. D. Ernst, “An experimental evaluation of continuous testing during development,” in *Proc. ISSA ’04*. ACM, 2004.
- [8] W. Choi, J. Brandt, and S. R. Klemmer, “Rehearse: Coding Interactively while Prototyping,” in *Extended Abstracts of UIST ’08*, 2008.
- [9] J. L. Snell, “Ahead-of-time Debugging, or Programming Not in the Dark,” in *Proc. Software Technology and Engineering Practice*, 1997.
- [10] A. Bangor, P. T. Kortum, and J. T. Miller, “An Empirical Evaluation of the System Usability Scale,” *Int. Journal of Human-Computer Interaction*, vol. 24, no. 6, 2008.