The Meager Validation of Live Programming

Johan Fabry
johan@raincodelabs.com
Raincode Labs
Brussels, Belgium

ABSTRACT

A key argument in favor of live programming languages and environments is that a better programming experience is achieved through maximizing feedback to the programmer. Intuitively, this is a compelling argument, but looking at validations of live programming languages and environments, the results are not encouraging at all. This essay concerns the question whether it is possible to validate that live programming is faster, or more correct, or that programmers have a better development experience. I provide a critical review of validations of live programming and talk about my own experience with such a user study. The goal of this text is to contrast the intuition with the scientific validation, such that we, as a community, can ask ourselves the question of whether we can validate the arguments in favor of live programming, and if so, how.

CCS CONCEPTS

• **Software and its engineering → Language types**.

KEYWORDS

Live Programming, User Studies, Validation

ACM Reference Format:
Johan Fabry. 2019. The Meager Validation of Live Programming. In *Companion of the 3rd International Conference on Art, Science, and Engineering of Programming (Programming '19), April 1–4, 2019, Genova, Italy*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3328433.3328457

Programming '19, April 1–4, 2019, Genova, Italy

ACM ISBN 978-1-4503-6257-3/19/04…$15.00
https://doi.org/10.1145/3328433.3328457

1 INTRODUCTION

In my current job I am sometimes confronted with the programming experience of present-day mainframe programmers, i.e. how the people that work on the software of our banks, insurance agencies, airlines, … produce and maintain code. Say that we want to develop a "Hello, World" in Cobol (of course), then the story goes a little something like this:

(1) We open our 'console window' that holds a connection to the mainframe, giving us a character-based interface of 80 by 35 characters.
(2) We create the file that will hold the program text. To do this, we run a special utility for file creation that has a text-based menu interface where we specify the name and various pieces of metadata.
(3) We open this file in the text editor, which is alike to vi, write the source code and save it.
(4) Compilation is a batch job that needs to be specified in a separate file, known as a JCL. So we create that JCL file using the file creation utility.
(5) In the text editor we write a batch job specification that lists the compiler command to run as well as the name of input and output file.
(6) Finally ready to compile, we submit the JCL to the operating system to execute whenever resources permit.
(7) To see the compilation job run, we open a job inspection utility that lists our current and past jobs, find our compilation job in the list and pull up the information on it. After the job has run, we can see the run completion status and the compilation output messages, if any. In case of errors, we can list the error messages, write them down somewhere and then go back to the editor to fix the errors
(8) To run the program, when successfully compiled, we need also use a JCL, i.e. repeat step 4 through 7. So we create the file, edit it, submit the job for execution, and after it finishes we can view its output in the job inspection utility.

Consider, in contrast, the development experience of writing "Hello, World" in a live programming environment such as Smalltalk:

(1) We open the development environment GUI.
(2) In a workspace window we enter one line of code.

(3) We select that line of code and from a context menu pick the 'run' option. If there is an error in compilation, the cause of the error is shown in the program text.
(4) Compilation and execution happens immediately and the result of execution is directly visible.

In terms of development experience, there are four steps to take instead of eleven, and moreover each step is arguably a more enjoyable programming experience. So, already with such a simple example, it seems obvious to state that programming in a live programming environment is a much better experience.

More fundamentally, the argument in favor of live programming languages and environments is that a better programming experience is achieved through maximizing feedback to the programmer [13]. Ideally this is done through a 'continuously active' system: every edit action triggers computation of the program and the display of computed values is updated live, as inputs vary. Such tightening of the feedback loop lightens the cognitive load of building accurate mental models of the system when its execution is observed, hence providing a better programming experience.

Compare the programming experience of mainframe programming versus the experience of live programming. The latter is "obviously" better. But is it better in a way that can be scientifically measured? Put differently, can we validate that live programming is faster, more correct, or that programmers have a better development experience? Intuitively, this is the case, but if we look at the work on validating live programming languages and environments this intuition does not hold up so well.

This essay provides a critical review of validations of live programming and talks about my own experience with such a user study. The goal of this text is to contrast the intuition with the scientific validation, such that we, as a community, can ask ourselves if live programming has been sufficiently validated, and if not, how to overcome the issues with the current validation.

2 THE MEANING OF LIVE

The field of live programming has been characterized by multiple authors in multiple ways, and not all of these characterizations coincide fully. I do not wish to take a stand here on a specific definition nor to introduce a new definition of live programming. However I consider it necessary to set the stage here for the argument I make in this text.

First I wish to distinguish between the terms of livecoding and live programming, and for this I refer to the characterization of Swift et. al. in their work on Visual Code Annotations [12]. In this text, the authors describe Livecoding as a performance art where the artist is a programmer that constructs an audiovisual art piece live in front of an audience. In contrast, live programming is the direct construction, manipulation and visualization of a program's runtime state.

The above definition of live programming allows for a significant level of flexibility in how direct construction is performed. In this respect, the liveness scale proposed in Tanimoto's work on Viva [13] is informative with regard to establishing a degree of liveness of a particular language. Even though the original levels only considered visual languages they can be recast in more general terms, as has been performed e.g. by Burnett et.al. [1].

For the purpose of this text, the four levels of liveness can be characterized as follows:

Level 1: Informative This is the base level, all programming languages are at least informative as their source code informs the eventual execution.
Level 2: Significant Is there a 1-to-1 relationship of the visible code with the running code? Changing a piece of the visible code should change its executing complement.
Level 3: Responsive Is updating done at every atomic edit operation, e.g. at every keystroke of a textual language?
Level 4: Live Is there a live visual representation of the code? Does it reveal (internal) program state or the program's actions in response to the input it receives?

Note that in non-visual languages the 4 levels of liveness do not necessarily impose a strict hierarchy: it is possible to have a level 4 visual representation of the running code without being responsive at level 3. An example of this is the Flogo II language [6].

In my opinion, the difference between level 2 and level 3 is a highly significant step in increasing the amount of liveness. This is because it marks the difference between needing to explicitly conclude all code editing actions versus edits automatically and immediately being integrated. In fact, Burnett et.al. only consider level 3 and 4 to be live programming [1]. Additionally, this is also hinted at by Hancock in his metaphor of live programming versus normal programming as the difference between using a water hose and a bow and arrow [6]. The continuous stream of water allows for much easier aiming at the target than the manual reloading required with a bow and arrow. In the same way, the continuous updating of program code makes it easier to establish whether the goal behavior has been reached. This is in contrast with needing to 'accept' every significant code edit action before the new behavior takes effect. Put differently, level 3 live programming minimises the "gulf of evaluation" [9] for establishing the effect of a code change, whereas level 2 requires an extra step to bridge this gulf.

Hence, when considering live programming, for the purposes of this text I only regard languages and environments of level 3 and above. Notably, Smalltalk and other dynamic languages are *not* level 3 since they need an explicit 'accept' or 'evaluate' action. Returning to our example in the introduction, we would hence imagine that level 3 and above languages should yield an even better programming experience than the Smalltalk example. However, in the case of "Hello, World" the example is arguably too simple to really show the advantages of live programming. This shows that in simple examples, the programming experience improvements of level 3 over level 2 would be almost trivial. So, the intuition is that advantages of a live programming environment will show themselves more clearly on complex cases.

3 THE PROOF OF THE PUDDING IS IN THE EATING

In my work on live programming I however found myself a bit disappointed with the aforementioned pudding. This is because, while there is an intuitive argument that shows that live programming is better than non-live programming, to the best of my knowledge there is no significant amount of validation of this assertion in the research literature. What I mean by significant is: a positive result of a user study or controlled experiment on a realistic development task, and considering both code writing and comprehension. I here give a brief overview of the work that comes the closest to such a validation of Level 3 or 4 languages and environments.

The work of Oney et.al. for expressing UI behavior in the InterState system [10] combines state machines and constraints in a level 3 and 4 live programming language. A comparative laboratory study was performed with 20 experienced programmers. The goal was to establish whether InterState code is more easy to understand and modify than event-callback code as used in traditional UI's. The study used two different tasks and two languages: JavaScript and InterState. Participants were asked to add a new behavior to an existing application and to make modifications to an existing behavior of a different application. The authors found that in both tasks participants were twice as fast when using InterState and also that: "Most participants felt comfortable with InterState's visual notation, calling it 'intuitive' and 'clean'." The setup of the experiment however focuses narrowly on evaluating an alternative solution to the arguably problematic event-callback model, not on benefits of live programming outside of this model. I argue that improvements are probably due to the issues of the event-callback model and not of the non-live nature of JavaScript. Returning to the study, we can also see that there is *no* data that shows whether InterState yields improvements in speed or accuracy for any kind of code understanding tasks. This lack of data is significant, since professionals spend at least half of their time on code understanding during code maintenance [4].

Forms/3 is a spreadsheet language [1] where, instead of using a matrix of cells, cells are freely placed in a *form* by the user. Each cell has a formula, and the computed value of a cell is the evaluation result of the formula. Wilcox et. al. used it to evaluate the advantages of live programming when considering debugging tasks [14]. A live version of Forms/3 is compared to a non-live version in an user study with 29 subjects working on two different tasks. All subjects worked with both the live and non-live version of Forms/3, one half first using the live version. The article is quite extensive and reveals a well-thought out and executed user study. I therefore consider the results quite important. They however do not shed a universally positive light on live programming. To quote part of the conclusion of the article: "liveness was not the debugging panacea that developers of direct-manipulation programming systems might like to believe it is" [14]. This statement is in line with an observation of the authors earlier in the text that they have "been unable to locate any studies of liveness's effects on either direct-manipulation programming systems or on debugging. However, there are results from related domains that seem to make assorted – and often conflicting – predictions about what can be expected about the effects of liveness on debugging in such programming systems." [14]. There are many interesting results in this study. For example, when considering quantitative results there was *no* statistically significant difference in debugging accuracy. In contrast, the qualitative results however show that 75% of subjects were more confident in their results and they believed liveness helped with accuracy. So the intuition that live programming is better is *not* backed up by the results of the experiment.

Krämer et.al. investigated how live programming languages impact the behavior of the developer [7] with a between-groups user study with 10 subjects using a live and non-live version of JavaScript. Subjects had to implement functionality in three different tasks where the challenges were of a different nature. I omit more details on the setup of the experiment here, and instead refer to the article, which provides a good overview of the work performed. An important result of this work is that the time necessary to complete the tasks was *not* significantly different between the live and non-live version of the language. The intuition that live programming is better is actually touched on by the text since authors state 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" [7]. Our intuition of being better may be correct, but maybe the improvement is negligible in many settings. One notable element that points in this direction is the following: the time to fix bugs that were introduced while coding was also measured in this

experiment. While there is no significant difference in the number of bugs introduced while developing, the time taken to fix them did *decrease* when using the live version of the language (in contrast to the results of Wilcox et. al.).

A last case I wish to present is the controlled experiments on the construction and code understanding of robotic behaviors in the context of the work of Campusano et.al. on Live Robot programming [3] (a paper of which I am a co-author). We performed two within-subjects controlled experiments, on 20 test subjects in total. The goal was to evaluate the accuracy and speed of development using live programming versus a classical means for robot behavior programming. The first experiment focused on program comprehension and the second experiment on program writing. In a nutshell, for this controlled experiment we found that live programming did *not* improve the programming experience in terms of speed and accuracy. This was so even though the test subjects liked it better than the non-live development environment. For the details on the setup and the results I refer to the text of the paper. Instead, here I wish to focus on some of the reasons why the experiment did not provide a good validation of live programming.

4 EXPERIMENTAL PRESSURES

To test our level 3 live programming language LRP [2, 5], a language for the programming of robotic behaviors, we set out to create a test setup that is as realistic as possible. The goal was to provide a thorough and in-depth validation of the overall programming experience of a live programming language. This was then to be followed by later on studying individual aspects of the language in detail, to see which ones turned out to be the most beneficial. (Sadly, those followups never happened.) To make our user study as realistic as possible, we set up a typical development environment using the standard tools for such a job. These includes the de-facto standard middleware for robotics, ROS [11] and its accompanying robot simulator, which is typically used for software testing in robotics[1]. Also, the programs to be worked on in the experiment corresponded to credible robotic tasks.

In retrospect, due to our goal of being as realistic as possible there were many elements that complicated the experiment and raise the question whether the benefits of liveness are testable in more complex examples. If so, the research question is which examples are big enough to be realistic and what is needed to be able to perform such tests.

I wish to focus more in detail here on four factors here that I consider the most important in restricting the feasibility of a realistic user study:

Time. Sessions of participants lasted up to four hours, which is arguably too long because, e.g., it could cause participant fatigue. We could however not shorten this time further without simplifying the tasks too much. This large amount of time was needed first because participants needed to perform two reasonably complex tasks: one in each language, and second because a warm-up phase was needed before each task to train the participants in the particularities of the language. Note that participants were already knowledgeable on robotics and had some experience with the typical development environment, i.e. middleware and simulator. Hence the warm-up phase was focused solely on the programming language used. Yet, even with taking this amount of time for each task our observations of the participants seem to indicate that the complexity of the task is low enough that it is straightforward to build a mental model. Hence liveness does not seem to provide an important benefit in this setting.

Unrealistic Code. In both the code comprehension and code writing experiment we had to use code that is arguably unrealistic. This is because even though the tasks and example code we provided were representative of robotic software, we had to obfuscate names. In the pilot studies we found that subjects were relying almost purely on the names of software entities to answer questions on comprehension and to incorporate them in their programs. Hence, to remove this bias caused by relying solely on names we renamed all entities to a four-letter random name, e.g. aomg, dzus, sphv. As a result, subjects needed to truly investigate the code to understand it and be able to reuse or adapt it. The code has however become more artificial due to this renaming, adding a confounding factor in understanding since the names are not realistic and probably also harder to remember. If the tasks were more complex and the provided code more intricate, naming would probably not be such a confounding factor. This since the name alone would not be sufficient to well understand the role of an entity. But, as said above, this was impossible due to the limitations on time.

Live All The Things. Even though the participants had experience with the robotic middleware that we used in our experiment, its API turned out to be an extremely large confounding factor. The API is inherently complex to work with, and this complexity impacted the entire development experience. We observed many errors being made in its use. Moreover, in almost every unfinished sub-task, the participants lost time due to issues using the external API. This is a notable factor because live programming languages do not live in isolation. If they are to be used on large or complex problems they will indubitably work with external resources, and we have seen that their nature may badly taint the live programming experience. We may need to 'Live all the things',

[1] No robots were harmed during the setup or execution of this experiment.

i.e. ensure that all significant parts of the system are live, before we can reap any noticeable benefits, and that is surely a high price to pay. Things may however not be so bleak: in our observations of the subjects, we found that there is a slight learning effect in the use of the API. Participants got more proficient in using it as time passed and hence its negative impact reduced slightly. Maybe more time working with the API could remove the barrier it poses, but then we end up in trouble with regards to the time needed to perform the experiment. Moreover, I am *not* convinced that the effect can be minimized sufficiently to allow the benefits of live programming to show, in part also because of the following:

The Force Of Habit. A last factor is the force of habit: it seems that since participants are not used to live programming they do not take advantage of all the benefits of liveness. One clear example of that is that in the program comprehension experiment almost none of the participants made any modifications to the program to see how the behavior changed. An example for writing code is the possibility to force the program to execute a certain behavior subroutine. This was useful in the experiment as it allowed skipping a lengthy sequence of steps that are not relevant to the code written by the subject. Yet almost none of the subjects made use of this shortcut, causing them to waste valuable time. A similar observation was also made by Kubelka et.al. [8]: developers are not taking advantages of the features offered by live programming. The consequence I draw of this is that with many obvious missed opportunities to use liveness, it is hard to evaluate the benefit of liveness. I however do not see a clear way to mitigate this issue. Using only participants that are used to the different features of the language being validated would introduce a bias in favor of it. To remove bias, all participants should be trained in both languages such that they have enough productive habits in both languages. I expect this however to be prohibitively expensive.

Conclusion. In our experiment to validate LRP in a realistic setting, we encountered multiple confounding factors that had an arguably negative impact on the results. It is however not clear how to remove these factors in a way that the setting remains realistic.

5 THE LAW OF DIMINISHING RETURNS

Yet why are these experiments so important? Quantitatively the evaluations of live programming are not convincing, but developers do report that they have a better development experience with live programming and this surely counts for something. I argue that this is however not sufficient, using the following counter-example: Mainframe developers that I have talked to are actually content with their development experience and argue that they are quite productive.

Development on the mainframe, as presented in the beginning of this text, seems horrid but in practice it turns out not to be the case. Mainframe programmers have a set of productive habits that allow them to navigate the variety of utilities in the flash of a few keystrokes. This is typically thanks to the use of UI macros provided by the application that connects to the mainframe. For example, creating a file barely takes more time then typing in its name, since setting its attributes takes a few key presses (if they are not already the default). Also, developers don't need to wait for compilation as these jobs typically are given top priority (since developer time is expensive). What we, as outsiders, see as overhead is 'just part of the process' and done quickly, out of habit, almost without thinking.

The point here is *not* to say that mainframe programming is as enjoyable or productive as live programming. But mainframe programmers being happy and quite productive (in their self-assessment) shows that we can *not* solely rely on qualitative results in user studies. We need quantitative results and I have not been able to find nor produce them.

From my experience, I fear that there is a law of diminishing returns here. Firstly, consider the rewards of using liveness: maybe level 3 is only marginally better than level 2, but at a very high cost of development of the language or system. Secondly, the same applies in evaluation: getting participants to be proficient with both systems is prohibitively expensive yet crucial to avoid them to miss opportunities. Yet it is not clear that these opportunities, which seasoned developers would make use of, can actually significantly improve the development experience. In summary, in both cases it seems that bigger investments are needed but that they yield smaller benefits; the law of diminishing returns.

So, while level 3 liveness may intuitively provide a better programming experience than level 2 or less, I fear that this is overshadowed by other factors, especially in the context of (the evaluation of) a larger workload. I have argued in this text that there is as yet no publication that refutes my fear. In order to be able to advocate live programming, the community should be able to show convincing validations in complex settings or be faced with the argument that the returns of level 3 live programming are too low to justify the investment.

ACKNOWLEDGMENTS

I am highly thankful to Miguel Campusano for the work on Live Robot Programming and for performing the bulk of the work on the user study I reflect on in this essay.

REFERENCES

[1] Margaret Burnett, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. 2001. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal*

of Functional Programming 11 (3 2001), 155–206. Issue 02.

[2] Miguel Campusano and Johan Fabry. 2017. Live Robot Programming: The Language, its Implementation, and Robot API Independence. *Science of Computer Programming* 133 (2017), 1 – 19.

[3] Miguel Campusano, Johan Fabry, and Alexandre Bergel. 2019. Live programming in practice: A controlled experiment on state machines for robotic behaviors. *Elsevier Information and Software Technology* (2019). https://doi.org/10.1016/j.infsof.2018.12.008 In Press.

[4] Thomas A Corbi. 1989. Program understanding: Challenge for the 1990s. *IBM Systems Journal* 28, 2 (1989), 294–306.

[5] Johan Fabry and Miguel Campusano. 2014. Live Robot Programming. In *Advances in Artificial Intelligence – IBERAMIA 2014 (Lecture Notes in Computer Science)*, Ana Bazzan and Karim Pichara (Eds.). Springer-Verlag, 445–456.

[6] Christopher Michael Hancock. 2003. *Real-time programming and the big ideas of computational literacy*. Ph.D. Dissertation. MIT, USA.

[7] Jan-Peter Kramer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2014. How live coding affects developers' coding behavior. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*. IEEE, 5–8.

[8] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. 2018. The Road to Live Programming: Insights from the Practice. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1090–1101.

[9] Donald Norman. 1988. *The Design of Everyday Things*. Doubleday.

[10] Stephen Oney, Brad Myers, and Joel Brandt. 2014. InterState: a language and environment for expressing interface behavior. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM, 263–272.

[11] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. 5.

[12] Ben Swift, Andrew Sorensen, Henry Gardner, and John Hosking. 2013. Visual code annotations for cyberphysical programming. In *Proceedings of the 1st International Workshop on Live Programming*. IEEE Press, 27–30.

[13] Steven Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing* 1, 2 (June 1990), 127–139. https://doi.org/10.1016/S1045-926X(05)80012-6

[14] Eric Wilcox, William Atwood, Margaret Burnett, Jonathan Cadiz, and Curtis Cook. 1997. Does continuous visual feedback aid debugging in direct-manipulation programming systems?. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*. ACM, 258–265.