Live Tuning:
Expanding Live Programming Benefits to Non-Programmers

Jun Kato Masataka Goto

National Institute of Advanced Industrial Science and Technology (AIST)
{jun.kato, m.goto}@aist.go.jp

Abstract

Live Programming allows programmers to gain information about the program continuously during its development. While it has been implemented in various integrated development environments (IDEs) for programmers, its interaction techniques such as slider widgets for continuous parameter tuning are comprehensible for people without any prior knowledge of programming and have been widely used for a long time. In this paper, we aim to introduce prior work on Live Programming research from the interaction point of view and relate it to Human-Computer Interaction research. We then name the subset of Live Programming interaction that only involves changes in constant values "Live Tuning." Our example IDEs that implement both Live Programming and Live Tuning interactions are showcased, followed by the discussion on the possible future direction of programming experience (PX) research.

Categories and Subject Descriptors H.5.2. **[Information interfaces and presentation (e.g., HCI)]** User Interfaces – GUI; D.2.6. **[Software Engineering]** Programming Environments – Integrated environments.

General Terms Design, Human Factors, Languages.

Keywords Live programming; live tuning; user interface; human-computer interaction; integrated development environment; programming language.

1. Introduction

Live Programming aims to eliminate the gap between the static source code and dynamic behavior of programs, providing continuous feedback of program content to programmers. It allows programmers to edit the program without halting its execution. The programming activity in an integrated development environment (IDE) can be divided into coding (defining new behavior), executing the program (observing the new behavior), and debugging it (repairing the behavior). Live Programming spans among all of these activities, and thus, is often considered as an effort to provide better Programming Experience (PX) that benefits programmers. It is certain that editing the code and immediately seeing the results is beneficial for programmers iterating the development process in exploratory programming. Though, is the benefit limited to programmers? What if we do a bit of forward-thinking?

Within Live Programming environments, programs are always live and editable. When the programs are developed in the web-based integrated development environments (WIDEs), the gap between the development and runtime environments can be eliminated. TouchDevelop [1] already achieves this in the web browser – anybody can pause the execution, navigates to the code editor and modify the appearance of the graphical applications. While TouchDevelop still requires explicit text-based programming, this implies potential of Live Programming techniques to be applied to end-user customization of applications.

To this end, we divide Live Programming experience into the operations that require prior knowledge of programming and the operations that do not require programming but just tuning parameters. While both operations immediately affect the behavior of applications, we name the latter operations "Live Tuning" and distinguish it from the fully-featured Live Programming experience (**Figure 1**).

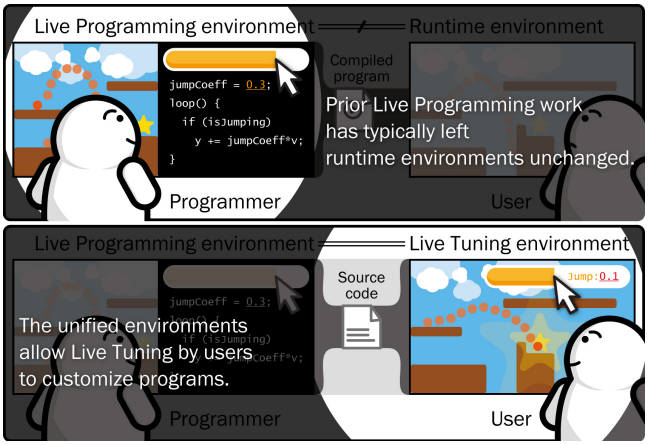

Figure 1. Live Tuning interaction based on Live Programming technique allows program customization by users.

Submitted to LIVE 2016.

Live Tuning does not provide full advantage of programming, but it helps people regardless of their knowledge of programming to customize the applications to meet their needs. Live Programming and Live Tuning can be technically supported by the same implementation but are just different interaction design. The rest of this paper introduces related work (**Figure 2**), defines Live Tuning, explains example implementations we made, and discusses the future research direction of Live Programming.

2. Related Work

2.1 Parameter Tuning in Live Programming

Parameter tuning interfaces can be widely found in Live Programming environments for improvisation of music and videos – especially visual programming environments such as vvvv [2] – and also for developing graphical applications such as Light Table prototype [3]. Text input is often not suitable for the continuous parameter tuning since keystrokes create jumps in values and the output gets dazzling. For instance, one keystroke can make the value ten times larger (by adding 0 to the end of an integer value) or smaller (by hitting the backspace and removing the last 0). The most typical interface for the purpose is a slider widget that allows modifying the corresponding constant numeric value without losing continuity. Physical sliders can also be used, as shown in **Juxtapose [4]** and lots of live coding environments. The sliders usually have the minimum and maximum values with fixed granularity of value range, but the slider interface can be extended to overcome such limitations such as elastic scrollbar [5] and zoom slider [6].

Text input and slider interfaces are so popular since most programming languages have string and numbers as primitive types. Although, "a set of knobs to control a hose's aim would be steady, but far less easy to work with [7]." Just as holding the hose with hands, direct manipulation is obviously more intuitive for changing positions and sizes of graphical objects. Such comfortable parameter tuning requires the development environment to be more domain-specific and have more knowledge about the applications. In the above case, a certain assumption is that applications are not for character-based user interfaces but have graphical outputs. If the applications deal with Color types, which is the case for most graphical applications, a color palette should be added beside the three independent sliders corresponding to R, G, and B values (e.g. Brackets [8] and other modern editors). **VisionSketch [9]** is a live programming environment for image processing applications that deal with computer vision algorithms. It is capable of graphically drawing shapes on input images as a means to specify a region of interest (yellow line in the figure) which is more intuitive than writing down the coordinate values by text. **SuperCollider [10]** has various extensions of user interfaces to fine-tune values used in programs that produce audio signals.

2.2 Testing Programs with Multiple Parameter Values

Even with intuitive user interfaces for continuous parameter tuning, showing a single output from the program is often not helpful enough to find the appropriate parameter value. For instance, if the program involves physical simulation or is a game and the parameter affects calculation for each frame, changes over time are more interesting than the last (or any other specific single) frame in the program execution history.

There are several ways to visualize changes over time. Stroboscopic visualizations overlay the frames rendering objects of interest with a certain transparency (Bret Victor's demonstration [11] and Light Table prototype [3]). Timeline interfaces cast time into x-axis and align frames in the horizontal direction. The interfaces can be found in many back-in-time debuggers such as Whyline [12] that correlates the line of code and the graphic object painted by the line. While ordinary back-in-time debuggers including Whyline do not provide Live Programming experience, **DejaVu [13]** is capable of updating the program output by re-executing programs with the recorded input. The print function of **YinYang [14]** correlates the line of code and the printed text. It provides a text-based console interface whose feature is comparable to the Timeline interface where one text line corresponds to one frame of interest.

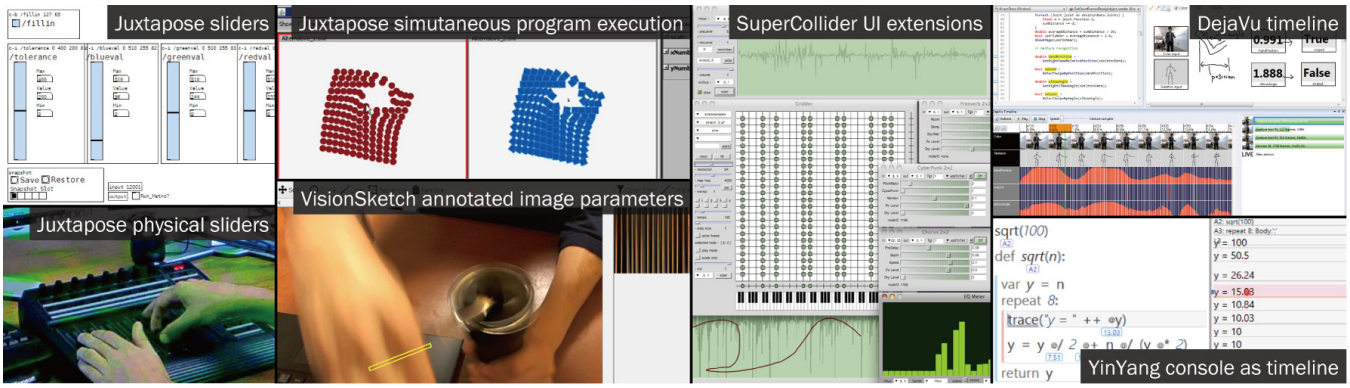

Figure 2. User interfaces for tuning parameters and visualizing program outputs. **Underlined bold related work** is included.

Beside time-coded program outputs, programmers are often interested in program outputs with multiple parameter values, too. Subtext [15] and Shiranui [16] allow the programmers to specify multiple test cases next to the program code, whose results are updated upon code edits. Subtext requires manual crafting of test cases. The programmers need to provide input values to the test functions by text input. In contrast, Shiranui and YinYang allow probing – which essentially allows the programmers to use the execution context around the specified (probed) line of code as input values to the testing functions. Furthermore, Shiranui allows to extract the probed context and define it as a new test case. Please note that all of the above examples target applications with the character-based output. To our knowledge, there is not yet a fully-featured Live Programming environment for graphical applications that allows defining multiple test cases whose results are simultaneously visualized. **Juxtapose [4]** is close to such instance in that it compiles multiple variations of the source code of graphical applications, executes the variations to create multiple windows, clone mouse and keyboard input events into each window, and allow simultaneously testing the variations.

3. Live Tuning

As introduced in the previous section, Live Programming environments often provide user interfaces for continuously tuning parameter values. From the perspective of interaction design, allowing users to tune parameters through sliders and other interfaces is observed not only in Live Programming environments but also in general end-user applications such as Photoshop whose filters can be configured with the slider interface. If the interfaces for parameter tuning are extracted from a Live Programming environments and all the other components for programming (such as the text-based code editor) are hidden from the user, do we still call it "a programming environment?" The answer is probably no, and thus, programming environments that only allows Live Tuning of parameters but live edits of logics (such as Juxtapose [4] and Unity [17]) cannot be called Live Programming environments. However, it does not necessarily mean that extracting the parameter tuning interfaces and exposing them to application users are useless.

We foresee that, in the age of web-based IDEs, programs will not only be edited but also be distributed and executed within the development environment. Some IDEs are already connected to the Internet, collecting usage information and improving its usability [18]. They will become platforms that cover the application lifecycle. As introduced in related work, the web version of TouchDevelop is one such example. It enhances the navigability to the code editor by recording the correlation between graphical outputs and the code element. The user can easily choose a graphical element while using the application, navigate to the corresponding code element, and edit the code. This workflow retains the fully-featured Live Programming experience, but it requires prior knowledge of programming to modify the behavior of the application.

Given these discussions, we propose to provide two levels of user interfaces in one IDE. While the one interface provides fully-featured Live Programming experience, the other limits the feature to parameter tuning and provides "Live Tuning" experience. From the end-user point of view, the proposed "Live Tuning" experience can be almost identical to the user experience of general applications. Although, the user interfaces for tuning parameter values are directly bound to specific code elements and allow deep customizations of the programs. The detailed discussion will be provided in the next section along with the brief introduction of IDEs that we implemented with both Live Programming and Live Tuning interfaces.

4. Example IDEs

With the direct correspondence between the "Live Tuning" interface and code elements, it gets straightforward and easy for the IDE to help programmers collect information on the application usage. For instance, analysis on parameter values provides knowledge on an appropriate range of the slider values. Such crowd-powered analysis is useful for making applications practical (e.g. visual design exploration [19]). It also enables the smooth transition from the Live Tuning interface to the Live Programming interface, lowering the barrier to begin programming.

To discuss further benefits of providing Live Tuning interfaces, we introduce two example IDEs as shown in **Figure 3**. These two IDEs corresponds to the following scenarios, respectively:

- Parameter tuning is too tedious to be solely handled by programmers (TextAlive)
- Benefits of Live Programming are desired for all users, not only programmers (f3.js)

4.1 TextAlive

TextAlive is an Integrated Design Environment that has two interlinked user interfaces for programmers and designers [20]. It is a Live Programming environment for developing programs that render Kinetic Typography (text animation) videos synchronized with songs in any time and display resolutions. The entire video strip is defined as a pure function of time. Programmers can create algorithms for animating text whose results are rendered on the canvas. Within TextAlive, populating Live Tuning interface is as easy as adding comments to variable declarations. Depending on the comment format, various kinds of Live Tuning interfaces

can be populated such as a color palette, slider, and button to start drawing freehand paths on the video canvas.

Programmers are good at abstracting the concept of text animation and write it down as code but are not necessarily good at choosing appropriate parameters. Designers are the ones good at such tasks, so TextAlive is equipped with the user interface for designers. It looks like a professional video editing software for them and allows intuitive authoring of Kinetic Typography videos. Within the user interface, there is a select box to assign algorithms for the animation to text components. Live Tuning interface appears here to allow editing parameters for the algorithms. With the Live Tuning interface, programmers and designers can easily collaborate on creating new media content.

4.2 f3.js

f3.js is an IDE for creating IoT devices [21]. It allows simultaneous development of hardware and software of the devices. Programmers can write a single piece of JavaScript code to define every aspect of the device. Just as a GUI development environment, f3.js is equipped with a code editor and interface builder. New sensor or actuator instances can be instantiated in the code and added to the interface builder that shows the development view of the device enclosure. Event listeners can be added to the instances, allowing to write event-driven code for controlling the microcontroller. Within f3.js, populating Live Tuning interface is as easy as adding comments to variable declarations. Depending on the type of their initial values, several kinds of Live Tuning interfaces can be populated such as a slider and checkbox.

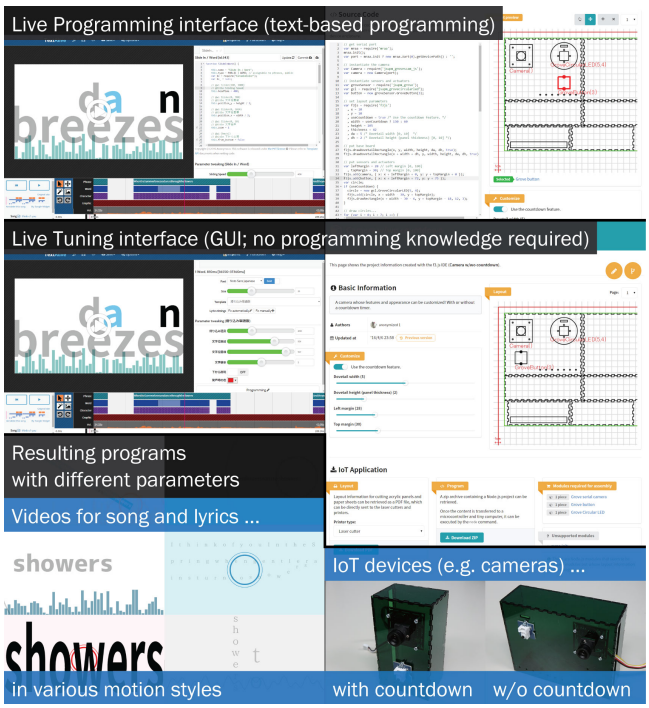

Figure 3. Screenshots of TextAlive (left; textalive.jp) and f3.js (right; f3js.org). Both are equipped with both Live Programming and Live Tuning interfaces.

Since the code defines both appearance and features of the IoT device, it is possible to declare a variable and change its value to produce variations of IoT devices. For instance, the code can contain the `useCountdown` Boolean variable declaration to control whether the enclosure contains space and holes for hosting a circular LED module or not as well as whether the microcontroller uses the LED module to countdown before capturing a photo or not. For this Boolean variable, it is not difficult to create two compiled variations (with and without the countdown feature) and let the user choose one. However, a single numeric variable can produce a number of compiled variations, and a combination of multiple variables further increases the number. Live Tuning interfaces eliminate the needs to precompile the information of the IoT devices, and thus, is essential to enable the end-user customization of the IoT devices.

5. Discussion

This section discusses the characteristics of Live Tuning interfaces in Live Programming environments and collects relevant research questions to be answered in the future work on programming experience (PX) research.

5.1 IDEs for Users with Varying Expertise

Live Tuning interfaces are **yet another layer of user interfaces** in Live Programming environments that allow users without prior knowledge of programming to customize programs. Existing tools with similar goals include on{X} [22] and IFTTT. Both of them have two levels of user interfaces for defining automated tasks. on{X} allows programmers to write JavaScript code and IFTTT allows to combine specific kinds of IF and THEN tasks. These created templates are then passed to people without prior knowledge of programming and customized with their parameter values to satisfy their needs.

Existing IDEs for Live Programming with similar goals to Live Tuning include vvvv [2] and VisionSketch [9]. Both of them are designed for programmers, but they intentionally separate user interfaces for visual operations (for constructing visual programming nodes and edges) and for text-based programming, providing similar separation of interactions for professional and novice users.

Other Live Programming environments can be easily extended to support Live Tuning interaction. In the TouchDevelop environment [1], Live Tuning can be implemented as support for direct manipulation of the GUI elements. Each graphical operation updates parameters in the text-based code defining positions, layouts, font sizes, and other graphical properties of the GUI elements. Threnoscope [23] is a Live Programming environment that visualizes its audio output as musical scores spreading concentrically. Again, Live

Tuning can be implemented as support for direct manipulation on the musical scores that edits the parameter values.

More general Live Programming environments can also be extended to expose the sliders for the users. Though, such feature is already provided by the "two-way (bi-directional) data binding" as seen in GUI development environments and toolkits such as Visual Studio and Windows Presentation Foundation (WPF; XAML + C#). As Live Tuning involves not only programmers but also program users, its effective implementation reflects what the users want to customize, often resulting in domain-specific IDE design. To catch their needs, Human-Computer Interaction point of view is crucial. What we showed in Section 4 are just two examples and we expect more to come in the near future.

5.2 Appropriate Flexibility to Edit Programs

Traditionally, to run programs, a "runtime" library is often needed that corresponds to a development environment such as "Visual C++ Runtime." The runtime library is a pre-compiled and degraded version of the development environment. There was **no flexibility** left to the user to edit programs.

Recently, web browsers are used as a unified platform for running various programs. Browser-based applications resolve the dependencies to external libraries by dynamically loading them without requiring any user operations. Thanks to reasonably fast Just-In-Time JavaScript compilers, the libraries are often distributed in the form of source code.

When these programs are developed in web-based IDEs (WIDEs), there is no gap between the runtime and development environments. The combination of Live Programming and WIDEs (e.g. TouchDevelop [1], TextAlive [20], and f3.js [21]) provides **maximum flexibility**, allowing to edit the program during its runtime without losing the context.

The vision of making every tangible software component editable in place has been long-awaited, and we believe that the combination has finally realized such dream. Though, we worry that making every part of the programs editable is not practical and can be even harmful. Novice users without a deep understanding of programs easily break the core functionalities of the programs and get confused. This is one reason why we consider there should be multiple levels of user interfaces that allow **varied flexibility** to edit the programs. Live Tuning is in-between "no flexibility" and "maximum flexibility" provided by the code editor. Discussion on the use of graphical interfaces (photos, videos, GUIs) as a means to customize programs can be found in our paper [24].

5.3 Eliminating More Kinds of Gaps for Better PX

Live Programming and other research such as Program Visualization have addressed the gap between the static representation and dynamic behavior of programs. WIDEs have addressed the gap between the runtime and development environments. There are, however, still other under-explored "gaps" in the program development process. Filling them is important to provide better Programming Experience (PX).

Local vs Remote **–** When programmers are developing programs that run on remote computers (e.g. robots, web applications), there is another gap between the development environment and deploying environment. The programs need to be transferred to the target computer before their execution. When there is only one programmer, it is a matter of latency. Otherwise, when there are multiple programmers making edits concurrently, an interaction design to support collaborative Live Programming is needed. There might be conflicts between edits made by the programmers.

Existing systems often discard the old edits and adopt the latest edits and implement efforts to prevent conflicts such as providing separate namespaces [25]. While more sophisticated conflict resolution can be implemented, conflicts in the context of collaborative Live Tuning pose a new research question: is it impossible to aggregate parameter values from clients and create a new value without annoying the users? Depending on the type of programs, interesting strategies are feasible such as 1) calculating the average or median of all client values, 2) passing the role among the clients at a certain interval, and 3) creating a poll for the best value. Some of these interaction has been implemented in Nightbird [26], a visual Live Programming environment for the improvisation of visual jockey performance.

Digital vs Physical **–** When programmers are developing programs with real-world input and output, there is a further gap between the computing environment and the real world. There is a latency for motors to arrive at the specified angle. Robots spend considerable time to bring things to the specified location. Heaters need time to heat up. 3D models take some time to print out. One certain way to address this gap is to implement a simulation of the real world. For instance, f3.js can be extended to simulate the physical properties of sensors and actuators. Then, programmers can write test cases such as occlusion detection that checks whether a camera module is placed at an appropriate location where other modules and enclosure of the device is out of the viewport.

It is, however, often difficult and time-consuming to implement practical simulation that resembles the real world behavior. In addition, the simulation does not necessarily provide "live feeling" to programmers. For instance, the usability of IoT devices cannot be checked by the 3D model on the display but only with its physical representation. To address such issue, we need to make significant progress in the research of programmable matter and radical atoms [27].

Furthermore, there will be programs dealing with various kinds of sensory data such as haptic sensation, taste, and smell. There is no such established representation of the sensory data that provides live feeling. This issue is critical but not specific to Live Programming anymore. Picode [28] tackles a similar problem of lack of intuitive representations

of human and robot postures. It extends a programming language to use photos as literals that represent posture data. Photos might also be helpful to represent other sensory data thanks to human's cross-modal ability (e.g. photos of flowers can represent their scent). Meanwhile, there is no one-to-one mapping between each photo and data. Some sensory data cannot be represented by any photo. There is neither linear correlation between photos and data. It is often impossible to compare photos and infer the differences between represented data. For more fluid programming experience, more investigation on live feeling is needed.

6. Conclusion

We coined the term "Live Tuning" that extends benefits of Live Programming to people without prior knowledge of programming by adding interactive user interfaces bound to source code elements. While Live Tuning does not allow directly editing program logics as Live Programming does, it eliminates the risk of breaking the core functionality of programs and yet enables their deep customizations. It can be implemented in any Live Programming systems but interesting implementations involve domain-specific knowledge on applications as shown in example IDEs (TextAlive for rendering videos and f3.js for controlling an IoT device and printing its enclosure). The discussion opens up broad range of future work which is not limited to but includes user interfaces for collaborative parameter tuning and intuitive representations of parameter values that provide "live feeling."

References

[1] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. "It's alive! continuous feedback in UI programming." In Proc. of PLDI '13, pp. 95-104.

[2] Meso group. "VVVV - a multipurpose toolkit." vvvv.org. 2009.

[3] Chris Ginger. "Connecting to your creation." www.chrisgranger.com/2012/02/26/connecting-to-your-creation, 2012.

[4] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. "Design as exploration: creating interface alternatives through parallel authoring and runtime tuning." In Proc. of UIST '08, pp. 91-100.

[5] Toshiyuki Masui, Kouichi Kashiwagi, and George R. Borden, IV. "Elastic graphical interfaces to precise data manipulation." In Proc. of CHI '95, pp. 143-144.

[6] Wolfgang Hürst, Georg Götz, and Philipp Jarvers. "Advanced user interfaces for dynamic video browsing." In Proc. of MM '04, pp. 742-743.

[7] Chris M. Hancock. "Real-time programming and the big ideas of computational literacy." PhD thesis, MIT, 2003.

[8] Adobe Systems Inc. "Brackets." brackets.io, 2012.

[9] Jun Kato and Takeo Igarashi. "VisionSketch: integrated support for example-centric programming of image processing applications." In Proc. of GI '14, pp. 115-122.

[10] James McCartney. "Rethinking the computer music language: SuperCollider." Computer Music Journal, 26(4), pp. 61-68.

[11] Bret Victor. "Inventing on Principle." Invited talk at CUSEC, Jan. 2012.

[12] Andrew J. Ko and Brad A. Myers. "Finding causes of program output with the Java Whyline." In Proc. of CHI '09, pp. 1569-1578.

[13] Jun Kato, Sean McDirmid, and Xiang Cao. "DejaVu: integrated support for developing interactive camera-based programs." In Proc. of UIST '12, pp. 189-196.

[14] Sean McDirmid. "Usable live programming." In Proc. of SPLASH Onward! 2013, pp. 53-62.

[15] Jonathan Edwards. "Example centric programming." In Proc. of OOPSLA '04, pp. 84-91.

[16] Tomoki Imai, Hidehiko Masuhara, and Tomoyuki Aotani. "Making live programming practical by bridging the gap between trial-and-error development and unit testing." In Companion Proc. of SPLASH '15 (poster), pp. 11-12.

[17] Unity Technologies. "Unity - Game Engine." unity3d.com, 2016.

[18] Marcel Bruch, Eric Bodden, Martin Monperrus, and Mira Mezini. "IDE 2.0: collective intelligence in software development." In Proc. of FoSER 2010, pp. 53-58.

[19] Yuki Koyama, Daisuke Sakamoto, and Takeo Igarashi. "Crowd-powered parameter analysis for visual design exploration." In Proc. of UIST '14, pp. 65-74.

[20] Jun Kato, Tomoyasu Nakano, and Masataka Goto. "TextAlive: integrated design environment for kinetic typography." In Proc. of CHI '15, pp. 3403-3412.

[21] Jun Kato and Masataka Goto. "f3.js – IoT apps with enclosures from single codebase." f3js.org. 2016.

[22] Microsoft. "on{X} - automate your life." www.onx.ms, 2012.

[23] Thor Magnusson. "The Threnoscope: a musical work for live coding performance." In Proc. LIVE '13 at ICSE.

[24] Jun Kato, Masataka Goto, and Takeo Igarashi. "Programming with Examples to Develop Data-intensive User Interfaces." IEEE Computer (Special issue on 21st Century User Interfaces), vol. 49, no. 7, Jul. 2016, pp. 34-42.

[25] Sang Won Lee and Georg Essl. "Communication, control, and state sharing in networked collaborative live coding." In Proc. of NIME '14, pp. 263-268.

[26] Yutaka Obuchi, Jun Kato, Masahiro Hamasaki, Masataka Goto, and Kentaro Fukuchi. "Nightbird Audience Node: adding audience support to VJ performance based on visual programming." github.com/FMS-Cat/nightbird.

[27] Hiroshi Ishii, Dávid Lakatos, Leonardo Bonanni, and Jean-Baptiste Labrune. "Radical atoms: beyond tangible bits, toward transformable materials." Interactions, 19(1), Jan. 2012, pp. 38-51.

[28] Jun Kato, Daisuke Sakamoto, and Takeo Igarashi. "Picode: inline photos representing posture data in source code." In Proc. of CHI '13, pp. 3097-310