Rethinking Programming "Environment"

Technical and Social Environment Design toward Convivial Computing

Jun Kato
jun.kato@aist.go.jp
National Institute of Advanced Industrial Science and Technology (AIST), Japan

Keisuke Shimakage
keisuke_shimakage@otonglass.jp
OTON GLASS, Inc., Japan

ABSTRACT

Computers have become ubiquitous in our life and work, and the way that they are built and used needs to be fundamentally improved. Most of the prior effort has been aimed at improving the programming experience for people with specific technical backgrounds (e.g., programmers, end-users, data scientists). In contrast, throughout this paper, we discuss how to make programming activities more inclusive and collaborative, involving people with diverse technical backgrounds. We rethink the programming environment from both technical and social perspectives.

First, we briefly introduce our previous technical effort in which the programming environment is shared between the developers and users of programs, eliminating the distinction between programming and runtime environments and fostering communication between them. Second, we introduce our social effort to support people who are visually impaired in implementing customized smart glasses that read words with a camera and speakers. We design their programming environment to consist of a software/hardware toolkit and engineers with domain expertise called "evangelists."

Learning from these experiences, we discuss several perspectives on convivial computing. To conclude, we argue that both technical innovations made on user interfaces for programming and understanding on the socio-technical aspect of domain-specific applications are critical for the future of programming environments, and accordingly, convivial computing.

CCS CONCEPTS

• **Software and its engineering** → **Integrated and visual development environments**; **Collaboration in software development**; • **Social and professional topics** → **Socio-technical systems**; • **Human-centered computing** → *Human computer interaction (HCI).*

KEYWORDS

convivial computing, programming environments, programming experience, live programming, social coding

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM Reference Format:
Jun Kato and Keisuke Shimakage. 2020. Rethinking Programming "Environment": Technical and Social Environment Design toward Convivial Computing. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3397537.3397544

1 INTRODUCTION

In 2011, Marc Andreessen noted "software is eating the world [2]." Computers have become more and more ubiquitous, and there are more and more programs running around us in our life and work. As a consequence, there are expectations for developers of programs to take more responsibility than ever. Within such a context, what is a programming environment, for whom is it, and how can we improve it to address this issue? This paper starts with this simple question.

When computer scientists refer to the term "programming environment," it usually means a computational environment in which a set of tools for developing programs is prepared for immediate use. A modern programming environment is often called an "integrated development environment" (IDE). It has not been called a "programmers' environment," and, therefore, nothing prevents it from being used by a wider variety of people. In a broader context, an environment is "the surroundings or conditions in which a person, animal, or plant lives or operates [25]" and could be designed to include non-computational artifacts, such as people who collaborate.

As we will introduce in the Background section, prior effort in the computer science-related research field has been mostly about making a programming environment more efficient for people with a specific technical background. The aim has typically been improving the overall programming experience for either programmers or end-users. As some programming activities heavily rely on domain specificity, there are recent and ongoing efforts to provide domain-specific programming experiences, for instance, for data scientists and artists. Regardless of this spectrum, from general to domain-oriented programming environments, environments are usually designed independently for people with a specific background.

On the contrary, as pointed out by Fischer, "although creative individuals are often thought of as working in isolation, the role of interaction and collaboration with other individuals is critical to creativity [9]." Through our effort, we aim at designing a programming environment in which people from diverse technical backgrounds gather and express their creativity. Such diversity is considered to be a vital source for social creativity, which was once

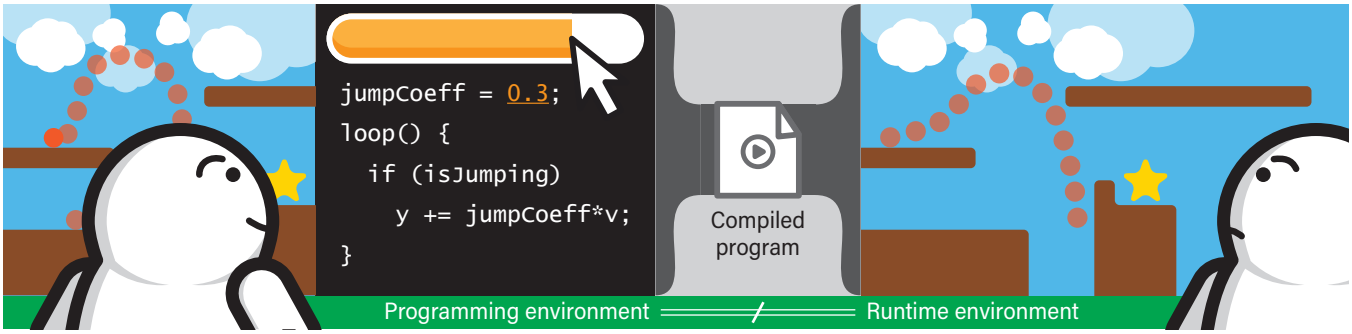

Figure 1: Conventionally, programming environments are usually separated from runtime environments. Users do not have access to source code and cannot edit programs to fit their needs.

examined in the context of end-user development [11] to realize "convivial" tools and systems as defined by Illich [13].

As concrete examples, we introduce our previous technical effort [15, 17] and ongoing social effort to rethink the programming environment to make it both more inclusive and collaborative. First, we introduce our previous technical effort in which the programming environment is shared between the developers and users of programs, eliminating the distinction between programming (buliding) and runtime (using) environments and fostering communication between them. Second, we introduce our social effort to support people who are visually impaired in implementing customized smart glasses that read words with a camera and speakers. We design their programming environment to consist of a software/hardware toolkit and engineers with domain expertise called "evangelists." Learning from these experiences, we discuss several perspectives on convivial computing. To conclude, we argue that both technical innovations made on user interfaces for programming and understanding on the socio-technical aspect of domain-specific applications are critical for the future of programming environments, and accordingly, convivial computing.

2 BACKGROUND

2.1 Respective Support for Programmers/End-users/Novices

To address the issue of there being a shortage of program developers, there have been prior research efforts toward supporting different kinds of developers. Roughly speaking, there are three areas. First, research on programming languages, software engineering, and human-computer interaction has developed general programming languages and environments to increase the productivity of existing programmers. Research projects such as Natural Programming [24] fall into this category. Second, most research on end-user programming has developed tools for people without prior knowledge of programming to take advantage of programming through domain-specific user interfaces. Recent relevant projects include End-user Software Engineering with a focus on testing and debugging experience [4]. Third, Computer Science for All [6] and other political efforts have taken place to increase the number of developers in the future. One can observe the growth of the research community in computer science education in the continually increasing number of accepted papers and attendees to the annual SIGCSE conferences [1].

These prior efforts improve the independent programming experience and typically do not affect how software programs are developed, distributed, and used. Programming environments are designed differently from runtime environments as shown in Figure 1, and the lifecycle of the programs is divided into build time and use time. Once programmers complete the development process in their programming environment and publish the program, users can run but edit it in their runtime environment.

In contrast, our approach is to implement a shared environment between programmers and users. It supports groups comprised of different kinds of people, not limited to programmers, to increase productivity as a whole, which Fischer defined as "social creativity [9]." The programs in such an environment are never considered as the final products but rather something unfinished that keep evolving in response to a variety of users' needs.

2.2 Domain-specific Programming Experience

As people use programs for various purposes, there is an increasing number of domain-specific programming languages and environments. The most notable recent examples include those for data scientists, including Jupyter Notebook (formerly named the "IPython Notebook" [26]) and other literate programming platforms, and one prior work investigated their usage [22]. Also, artists write programs for artistic purposes. Live coding is an activity in which musical and graphical performances are improvised in front of an audience while showing the code editor to them [7]. Generative design is an iterative process in which artists generate a variety of design alternatives using programs, choose desirable parameters, and look for a favorable design. Example systems that support generative design include Houdini [29], TextAlive [19], and Para [14].

In contrast to conventional programming environments that put much focus on text-based user interfaces, these domain-specific programming environments need to handle more data-intensive workflows and come with more graphical user interfaces [18]. We follow the same path of improving user interfaces of programming environments but for supporting communication between programmers and non-programmers.

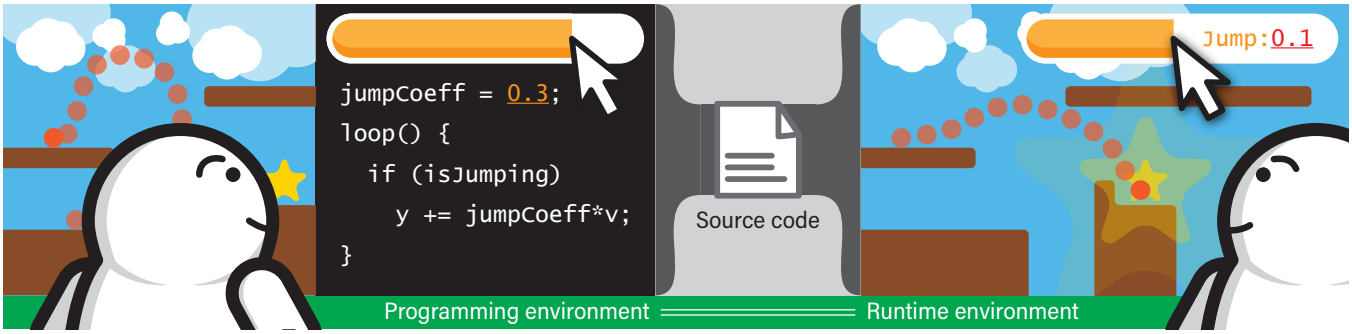

Figure 2: In our technical effort, we designed runtime environment that is same as programming environment except for its dedicated graphical user interfaces.

2.3 Live Programming

Live programming is an emerging paradigm for programming environments that enables interactive edits of programs without losing the context. Since it allows for fluid mode switch between the building and use of programs, it can be a technical platform for distributing applications on which the user can seamlessly become a developer. TouchDevelop [3] was one such example that allows the user to pause any running program, navigate to relevant pieces of code, and edit them. Scratch on the Web provides a similar experience to "remixing" existing programs [12]. Codestrates [27] is a web-based literate programming environment with support for real-time collaboration, enabling live programming of web applications.

Live programming environments have the potential to eliminate the gap between the user and developer technically. However, they still require the user to have the same level of expertise in programming as the original program developer. Therefore, the user still needs to spend time on learning programming to benefit from such an underlying technical platform. We will introduce relevant interaction design to expand the benefits of live programming to non-programmers.

3 RUNTIME ENVIRONMENT AS PROGRAMMING ENVIRONMENT

A runtime environment, in general, refers to an environment in which users run built programs, and it is different from a programming environment. A runtime environment cannot be used to modify program specifications, and the users need to ask for the developers' help. The programming environment is not for the users, and the developers need to leave it to collect feedback from the users.

These days, social coding platforms and bug reporting services support communication and fill the gap between the two environments. However, since users are running optimized programs without much information for debugging, reported issues often lack context information, and to provide context information, the users need to follow a tedious process.

To address the issue, we considered providing a full-fledged programming environment to users with dedicated user interfaces for them, as shown in Figure 2. The following subsections introduce its use for collaboration and communication. A web-based programming environment for building Internet of Things devices with a piece of JavaScript code, named "f3.js [16]," is used as an example that implements the proposed interaction design.

3.1 Live Tuning

Live programming environments usually show an ordinary text-based code editor to the programmer, requiring the user to have prior knowledge of programming to a certain extent. Meanwhile, our prior study [21] suggests that augmenting the code editor with graphical representations would even help end-users to comprehend and edit program behaviors.

Live Tuning [15] is an interaction design for live programming environments that hides the code editor but shows sliders and other appropriate graphical user interfaces for editing part of the running programs. While the code editor exposes everything about the program specifications, graphical user interfaces limit the freedom of editing a program to a certain extent to prevent the user from breaking the core specifications. Live Tuning implements a multi-layer interface design [28], one of which is a base layer for the developer, and the other is built for the user. It is also an application of the meta-design framework [10], in which a meta-designer (developer) defines degrees of freedom for the user with which the user can customize the software to fit their needs.

3.2 User-Generated Variables

In Live Tuning, the developer and user of programs use the same programming environment to run the programs. They can manipulate graphical user interfaces to edit variable values to customize programs. The customized programs are indeed the results of collaboration between the developer and user. However, this is unidirectional in that the developer defines everything, and there is no support for the user to provide feedback.

User-Generated Variables [17] is an interaction design built on top of Live Tuning. It further allows users of programs to declare new variables and add mock user interfaces bound to the variables. When a new variable is declared, the programming environment notifies the program developer via social networking services such as Twitter. Then, the developer can edit the code to remove the mock user interfaces or to make it work by implementing a logic that binds the variable values to the behavior of the program.

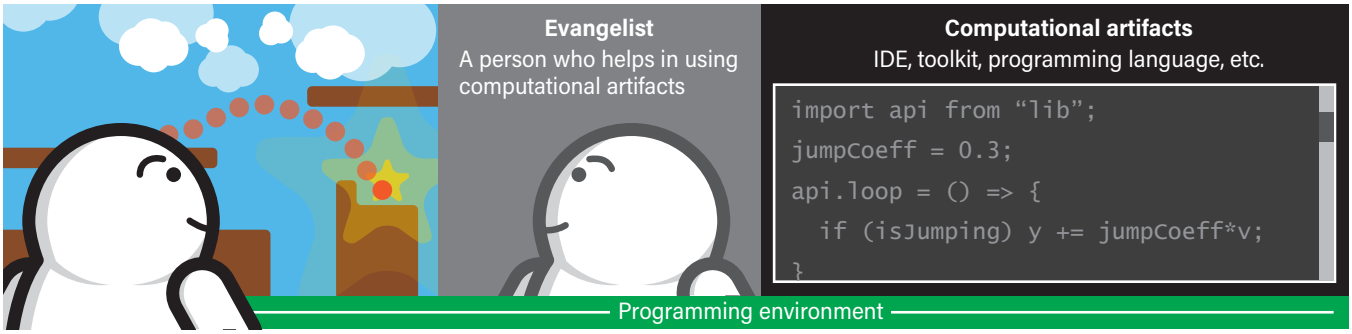

Figure 3: As social effort, we designed programming environment consisting of evangelist and computational artifacts.

3.3 Technical Efforts toward Convivial Environment

These technical efforts do not only eliminate the difference between programming (building) and runtime (using) environments but also make programming more inclusive. Through the introduction of Live Tuning, a programming environment becomes not only for those with prior knowledge of programming but also for users of programs. Through the introduction of User-Generated Variables, a programming concept (the variable) becomes a medium for mutual communication between them.

When tools for programming have built-in features for collaboration and communication, the programming environment can be considered to include not only computational artifacts but also people who collaborate and communicate with each other. In such a circumstance, the technical effort cannot stand on its own, and we need to think of the community of people instead of a person using the environment. The next section introduces our ongoing effort to think of such a social aspect.

4 EVANGELIST AS PART OF PROGRAMMING ENVIRONMENT

One benefit of an environment design that consists of computational artifacts resides in its scalability. Once the design is fixed, a programming language, a library, a toolkit, or a programming environment can be distributed through the Internet and can be used by anyone in theory. However, there has always been someone who has an urgent need for the provided functionality, but the functionality is not accessible.

One notable example is the use of computers by people with impairments. In particular, hardware devices for these people to deal with their impairments face various requirements, and existing products often fall short of meeting the needs of individuals and their particular symptoms. While they need personalized devices, they often cannot customize devices by themselves. As demonstrated in the previous section, f3.js [16] is a programming environment that can be used by both a programmer and non-programmer. However, it is primarily designed for those without any impairments.

In this section, we report the development process of a toolkit named "OTON GLASS" for building smart glasses that help a visually impaired person to read words. Throughout the development, we have been implementing a programming environment centered on a visually impaired person, or a "maker," who wants to build customized smart glasses, as shown in Figure 3. The toolkit part consists of a set of hardware blueprints that can be 3D printed and software components that can be freely customized. Meanwhile, we have formalized the environment to contain not only such computational artifacts but also a person who can help them to achieve this goal, whom we call an "evangelist." As case studies, we asked three teams of toolkit evangelists and users to develop their glasses with unique shapes and features.

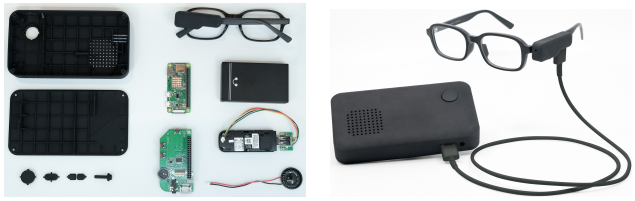

Figure 4: Overview of hardware components of OTON GLASS and its assembled form.

4.1 OTON GLASS

"Oton" is a Japanese informal term for "father." OTON GLASS was initially developed as a personal device in the form of glasses to help the father of one of the authors who began suffering from a macular disease in 2012. Then, we got to know people who had similar issues, and tens of copies of the device became available for a limited number of testers. Later, we noticed that visually impaired people had a variety of symptoms, and more importantly, they had their own style of wearing glasses.

OTON GLASS is composed of 3D printed enclosures, glasses, a camera module, a Raspberry Pi module, a custom PCB module, a battery module, an LTE module, and a speaker module. Since the components are off-the-shelf, it looks straight-forward enough for each person to customize their own device. However, on the one hand, it was not realistic for we (a small team of people) to respond to all of the customization requests, and, on the other hand, it was also not feasible for people who are visually impaired to customize and build the device by themselves, even with the help of an ordinary toolkit or a programming environment such as f3.js [16].

Figure 5: OTON GLASS toolkit allowed three teams of makers and evangelists to develop devices with wide variety of shapes and features.

As a consequence, we decided to design a toolkit that is used by people with two distinct roles. One role, called "maker," is to design, build, and use the customized device. The other role, called "evangelist," is to understand how to use the toolkit and help to build the customized device with the toolkit. The evangelist serves as a sort of catalyst to make the design process happen with the help of the toolkit. We call this pair of toolkit and evangelist "a programming environment" for the maker.

4.2 Case Studies

To investigate the effectiveness of the programming environment design, we asked three teams of makers and toolkit evangelists to develop their glasses (Table 1). The development period was from July to August 2019, and photos of the outcome from these case studies are shown in Figure 5. The makers had diverse backgrounds, and those with visual impairments are marked as *(VI)*. Snowball sampling was used to invite all of the makers.

The evangelists were those who had gotten involved in the development of the OTON GLASSby the time when the case studies started. Each of them had different types of expertise – one engineer was capable of both software and hardware design (referred to as *SH*), another was capable of hardware design (referred to as *H*), and the other was capable of software design (referred to as *S*). For each team, all of these evangelists could have been potentially available. Those who contributed to the outcome are credited in the table.

Maker(s)	Outcome	Evangelist(s)
Architect (VI) & interaction designer	OTON GLASS in a detachable pen form	SH, H
Paralympian (VI) & his helper	Real-time communication feature	S
Two software engineers (both VI)	API wrapper and its three applications	SH

Table 1: Three teams and their outcome using toolkit.

4.2.1 Case of Architect (VI) & Interaction Designer. One architect had a progressive disease that gradually made it difficult to see things. When he found something to read, he needed to use a smartphone to take a picture of it and perform a pinch gesture to magnify the content. For instance, there are various kinds of remote controllers, and since they have unique button layouts, the camera roll of his smartphone contained many pictures of the user interfaces of the remotes. The architect teamed up with an interaction engineer, met the evangelists (including SH and H), and had a brainstorming session to think of how to personalize the OTON GLASS for himself.

In the end, the architect came up with the idea of customizing the appearance of the OTON GLASS. He uses a pen for his professional work all the time and feels a bit lost when he does not have the pen. Therefore, what if he could transform the OTON GLASS to have a pen form? In its original 3D model, the camera module was always attached to the temple of the glasses. Since the architect and the interaction engineer did not have much expertise in hardware modeling, the evangelists (SH and H) helped them to create a 3D model into which the camera module is built.

As a result, the architect could hold the OTON GLASS camera in his hand. Since the pen was a particularly personal object for the architect, he could have a more embodied feeling with the OTON GLASS device. For instance, when he was not holding it, he could put it behind his ear, which was a familiar action to him.

He also appreciated that the read-aloud feature of the OTON GLASS got more accessible and thus practical. Using this feature used to require finding targets carefully, but the pen form enabled quick zapping among potential targets. Through such experience, he started to feel that his body was augmented and extended and commented that he wanted to use the device longer.

4.2.2 Case of Paralympian (VI) & His Helper. A visually impaired Paralympian and a professional supporter who accompanied the

Paralympian on exhibition matches teamed up in the second case study. When the Paralympian goes to an exhibition match, it is often the case that the number of accompanied professional supporters is smaller than the number of Paralympians, so there arises the need for remote supporters.

While the original OTON GLASS could already help the Paralympians, the features were limited to recognizing and reading textual information. Examples of further difficulties faced by the Paralympians included, but were not limited to, 1) choosing a favorable food option from a restaurant menu, 2) finding their record from the official score table, 3) getting directions to their hotel room with the right room number, and 4) choosing the uniform clothes that were in the right color.

The concept-making of complementing the OTON GLASS features with remote supporters went in a straight-forward manner, and for the actual implementation, evangelist S contributed enormously toward the working prototype. In the end, a voice chat feature using the Web Real-Time Communication (WebRTC) was added to OTON GLASS and could be used practically by the team members.

Unlike ordinary teleconference applications such as Skype, utilizing a camera attached to glasses means directly sharing what one sees. This worked quite well for the team and would also work in a community of people with a stable trust relationship.

4.2.3 Case of Software Engineers (VI). The third case study was a team of two software engineers. One engineer was blind, and the other had a slight issue with his eyesight. The blind engineer had developed software by touch-typing a keyboard with auditory feedback but had difficulty, especially when he needed to build up mathematically complex algorithms. During the development, the other engineer helped him on such occasions. Evangelist SH helped these two engineers to modify the OTON GLASSsoftware.

The blind engineer had his own view on programming and desired a personalized application programming interface (API) of the OTON GLASS software for himself. As such, the team first developed an API wrapper around the existing OTON GLASS feature as well as the other features provided by the Raspberry Pi device. With the help of the evangelist, the team transformed the OTON GLASS toolkit into another toolkit with its own distinct API.

To demonstrate its usage, they prototyped three applications. The first application intercepts the OTON GLASS feature of reading recognized text and allows the user to navigate to a specific location in the text with the buttons of the OTON GLASS hardware to skip or rewind content being read aloud. The second application also intercepts this feature and allows photos to be shared with his friends instead of sending them to the OCR API. The third application completely removes the OTON GLASS features and utilizes the buttons to control music playback.

4.3 Social Efforts toward a Convivial Environment

The case studies demonstrated the power of the combination of evangelists and makers. Each case was successful and unique to the maker in each team, who had different symptoms and lived their life differently. The makers could not implement a wide range of applications without the help of the evangelists. Technology-oriented toolkits that do not require support from a community of people would scale better, but to solve socio-technical issues in the wild, connecting people with appropriate knowledge would be beneficial, recalling the discussion on social creativity [9].

Also, the fact that three teams in the case studies could work in parallel in the same development period implies that the toolkit-evangelist-maker model would scale better than the conventional developer-maker (tailor-made) model. If there were no toolkit, each team would have needed to create the product from the ground up, which would not have been feasible. We would love to see more toolkits developed with this balanced approach.

5 DISCUSSIONS AND FUTURE WORK

5.1 Rethink Programming "Environment"

There has been increasing interest in improving programming environments among multiple research fields. Researchers in the programming language field are thinking of programming experience design beyond language design, and relevant workshops such as PX (Programming Experience Workshop) and LIVE (Workshop on Live Programming Systems) were born as a result. A programming environment provides a programming experience, and this same phenomenon has been observed from different perspectives. It has become common for researchers in the human-computer interaction field to design integrated development environments and their plug-ins, going beyond designing a single toolkit or user interface for programming. An environment is a set of multiple tools designed for a particular workflow of application development.

Following this interdisciplinary literature, we aimed at providing new perspectives on programming environment design by considering the communicative and collaborative aspects of environments. To do so, we introduced two concrete examples of programming environment design. First, we designed a programming environment that is not monolithic nor only for those with expertise in programming but contains flexible multi-layered user interfaces. The environment provides appropriate user interfaces and helps them communicate with people from diverse technical backgrounds. Second, we designed a programming environment that consists of not only computational artifacts but also people who support programming. Throughout these examples, we showed that a programming environment designer could design how a community of people communicate and collaborate within the environment. This is no longer a purely technical effort and involves social effort. Future work could elaborate on the toolkit-evangelist-maker model and investigate social programming-environment design that further accelerates social creativity.

5.2 Scalability and Social Inclusion

Technical efforts to improve programming environments are scalable in that the implemented features work automatically. At the same time, however, there could be someone who is out of the scope and needs dedicated help to use a feature. Social efforts could complement technical efforts and address such limitations. For instance, f3.js [16] is a web-based programming environment and can run on any modern web browsers. It allows the user to output customized microcontroller firmware and a PDF file to print

with a laser cutter. However, visually impaired people might have difficulties in assembling the physical computing device. OTON GLASS addresses this by introducing an evangelist of the toolkit.

We need to be mindful that the scalability provided by the technical implementations is not the goal by itself. We consider the scalability to be beneficial since it contributes to making programming environments more inclusive. Nonetheless, we should not stop our technical efforts since we could sometimes replace social efforts with novel techniques. For instance, we might be able to realize computational support for visually impaired people to aid them in building a physical device by themselves, potentially eliminating the need for them to gain help from an evangelist.

We designed both of the two example programming environments with domain-specific insights gained from our expertise and dialogues with users. In particular, f3.js was designed in the context of physical computing literature, and OTON GLASS was designed in the context of DIY-AT (do-it-yourself assistive technology) literature. They pragmatically provided state-of-the-art solutions to real issues that the users were facing. Environment design needs to carefully consider domain specificity to seek the right balance between scalability and social inclusion.

5.3 Programming as Communication

The example programming environments open up new pathways for users with diverse technical backgrounds to communicate with each other through programming-related activities. We envision that programming should be a more social activity than what it is right now. While existing efforts to make the software development process social are often called "social coding," it is mostly about developing software with the help of an optional social activity and not the opposite (developing software naturally being a social activity).

In a section titled "Design as Communication" in a book written in 1995, Thomas Erickson wrote, "It is useful to think about design as a process of communication among various audiences. Central to this discussion is the notion of design artifacts, material or informational objects that are constructed during the design process [8]." We should consider what programming environments can do when we replace "design" with "programming" and "design artifacts" with "programs" in the quote and how the process can involve "various audiences."

One notable relevant work would be Picode [21], a programming environment that integrates the activity of taking photos into the programming workflow. Within the environment, a photo is a reference to a constant value of an integer array representing the posture data of a human or a robot. Taking a photo defines a new constant value and is part of the programming activity. In workshop events utilizing the Picode environment, the photo-taking activity was often an opportunity for the participants to communicate and discuss how the program should behave.

"Programming as Communication" like this example is yet much under-explored, and there seem to be many opportunities for future work. For instance, just as the static value stored in a variable was used as a communication medium [15, 17], future work could investigate the use of function as a communication medium. A user without prior knowledge of programming could demonstrate the desired behavior to the programming environment, which would generate a mock function definition, and another user would be notified and could implement the actual function body. This is similar to programming by demonstration, except for the subject inferring the function definition being a human user instead of a system observing the demonstration.

5.4 Empowerment through Programming

Technologies for personal fabrication have enabled the fabrication of personalized products, and such applications that help people with impairments are called "DIY-AT" (do-it-yourself assistive technology). However, it is still difficult for people with an impairment to become "makers" by themselves, and in many cases, their parents, friends, and care-givers utilize DIY-AT technologies. To this end, one prior work in which a personal fabrication workshop was held for people with disabilities revealed that they could experience the feeling of empowerment through a do-it-yourself activity in which physical objects were built with the help of workshop facilitators [23].

The prior DIY-AT research has primarily focused on building passive physical objects, and our work has focused on building interactive computational artifacts. The state-of-the-art in both kinds of literature aims at allowing diverse people to get involved in the building process, and future work should investigate the characteristics of empowerment achieved through the involvement. In particular, we are interested in the spectrum of empowerment provided by different levels of involvement in the activities of programming. Conventionally, a limited number of people completed programming by themselves, and our work revealed that there are more ways to get involved in the process, such as tuning parameters, declaring variables, and doing so with the help of evangelists. We should probably also look into the research of the open-source community and connect these pieces of literature.

5.5 Remote and Local Collaborations in Programming

A prior discussion on social creativity [9] pointed out that the distance across the spatial dimension is one of the critical factors for collaborative creative output. Remote collaboration is often criticized for being less productive than local collaboration. However, it is said to highlight shared concerns rather than shared locations. Since it allows more people to join the collaborative process, their local knowledge can be more easily exploited, potentially resulting in more creative output. To follow this insight, we implemented f3.js [16] as a web-based programming environment. Communication and collaboration in the environment happen completely remotely, just as they happen successfully in open-source communities.

In contrast, in the OTON GLASS project, the locality of people played an essential role. Because of the physical nature of the development of smart glasses, and probably also because of the visual impairment of the people involved, efficient development could happen only where the people were physically present. The term "community-based prototyping" was coined by those in the OpenPool project [20] through their collaborative experience of developing a digitally-augmented billiard table. They also pointed out

that the development of physical devices often requires the participants to be there, forcing them to meet face-to-face and encourage new audiences to participate in the process.

The locality of products and data was discussed in the FabCity white paper [5]: "It (FabCity) is a new urban model of transforming and shaping cities that shifts how they source and use materials from 'Products In Trash Out' (PITO) to 'Data In Data Out' (DIDO)." Our findings from the OTON GLASS project are two-fold. First, a small number of people (such as those who are visually impaired) cannot always be supported sufficiently by the DIDO model. Second, evangelists who have specific know-how that is difficult to be transferred as digital data could potentially address technical issues. We foresee that people will serve as the carrier of complex information, and "People In People Out" (PIPO) could augment the DIDO model. It is also possible that technological advancements in virtual reality, augmented reality, and robotics will make remote collaboration more efficient, making such a PIPO process completely virtual.

6 CONCLUSION

Throughout this paper, we aimed at extending the design process of a programming environment to consider its communicative and collaborative roles. We introduced two concrete examples of programming environment design, with a focus on technical innovation and social inclusion, respectively. While technical innovations made on user interfaces for programming make the programming environment scalable and invite more people, understanding the socio-technical aspect of domain-specific applications and the accompanying pragmatic social effort is essential to making the environment inclusive. We argue that more work on programming environment design with the right balance between these two perspectives is vital for the future of convivial computing.

ACKNOWLEDGEMENTS

We sincerely thank all the collaborators who contributed to the research and development of the example systems and their interaction design introduced in the paper, most notably Masataka Goto for f3.js [16], Live Tuning [15], and User-Generated Variables [17] and Keita Miyashita and Tomoaki Sano for OTON GLASS. This series of work was supported in part by JST ACCEL Grant Number JPMJAC1602, Japan. We also thank Satoshi Sekiguchi, who triggered this research through a thought-provoking comment on the distinction between development and runtime environments. At the Convivial Computing Salon, Philip Tchernavskij kindly provided an informative and constructive critique, which should be accessible on our project page[1]. Last but not least, we thank all users of the programming environments we have built.

REFERENCES

[1] 2019. *SIGCSE '19: Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA). Association for Computing Machinery, New York, NY, USA.

[2] Marc Andreessen. 2011. Why software is eating the world. *Wall Street Journal* 20, 2011 (2011), C2.

[3] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. It's Alive! Continuous Feedback in UI Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 95–104. https://doi.org/10.1145/2491956.2462170

[4] Margaret M. Burnett and Brad A. Myers. 2014. Future of End-User Software Engineering: Beyond the Silos. In *Proceedings of the on Future of Software Engineering* (Hyderabad, India) *(FOSE 2014)*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/2593882.2593896

[5] Fab City. 2018. Fab city whitepaper: Locally productive, globally connected self-sufficient cities. Website. Retrieved January 16, 2020 from https://fab.city/uploads/whitepaper.pdf.

[6] CSforALL. [n.d.]. CSforALL: Computer Science for ALL Students. Website. Retrieved January 16, 2020 from https://www.csforall.org.

[7] Roger T. Dean and Alex McLean. 2018. *The Oxford Handbook of Algorithmic Music*. Oxford University Press. https://www.oxfordhandbooks.com/view/10.1093/oxfordhb/9780190226992.001.0001/oxfordhb-9780190226992

[8] Thomas Erickson. 1995. *Notes on Design Practice: Stories and Prototypes as Catalysts for Communication*. John Wiley & Sons, Inc., USA, 37–58. http://www.pliant.org/personal/Tom_Erickson/Stories.html

[9] Gerhard Fischer. 2005. Distances and Diversity: Sources for Social Creativity. In *Proceedings of the 5th Conference on Creativity & Cognition* (London, United Kingdom) *(C&C '05)*. Association for Computing Machinery, New York, NY, USA, 128–136. https://doi.org/10.1145/1056224.1056243

[10] Gerhard Fischer, Daniela Fogli, and Antonio Piccinno. 2017. Revisiting and broadening the meta-design framework for end-user development. In *New perspectives in end-user development*. Springer, 61–97.

[11] Gerhard Fischer and Andreas Girgensohn. 1990. End-User Modifiability in Design Environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, USA) *(CHI '90)*. Association for Computing Machinery, New York, NY, USA, 183–192. https://doi.org/10.1145/97243.97272

[12] Benjamin Mako Hill and Andrés Monroy-Hernández. 2013. The Cost of Collaboration for Code and Art: Evidence from a Remixing Community. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work* (San Antonio, Texas, USA) *(CSCW '13)*. Association for Computing Machinery, New York, NY, USA, 1035–1046. https://doi.org/10.1145/2441776.2441893

[13] Ivan Illich and Anne Lang. 1973. *Tools for conviviality*. Harper & Row New York.

[14] Jennifer Jacobs, Sumit Gogia, Radomír Mundefinedch, and Joel R. Brandt. 2017. Supporting Expressive Procedural Art Creation through Direct Manipulation. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 6330–6341. https://doi.org/10.1145/3025453.3025927

[15] Jun Kato and Masataka Goto. 2016. Live Tuning: Expanding Live Programming Benefits to Non-Programmers. In *Proceedings of the Second Workshop on Live Programming Systems* (Rome, Italy) *(LIVE '16)*. 6. https://junkato.jp/publications/live2016-kato-livetuning.pdf

[16] Jun Kato and Masataka Goto. 2017. F3.Js: A Parametric Design Tool for Physical Computing Devices for Both Interaction Designers and End-users. In *Proceedings of the 2017 Conference on Designing Interactive Systems* (Edinburgh, United Kingdom) *(DIS '17)*. ACM, New York, NY, USA, 1099–1110. https://doi.org/10.1145/3064663.3064681

[17] Jun Kato and Masataka Goto. 2017. User-Generated Variables: Streamlined Interaction Design for Feature Requests and Implementations. In *Companion to the First International Conference on the Art, Science and Engineering of Programming* (Brussels, Belgium) *(PX '17)*. ACM, New York, NY, USA, Article 28, 7 pages. https://doi.org/10.1145/3079368.3079403

[18] Jun Kato, Takeo Igarashi, and Masataka Goto. 2016. Programming with Examples to Develop Data-Intensive User Interfaces. *Computer* 49, 7 (7 2016), 34–42. https://doi.org/10.1109/MC.2016.217

[19] Jun Kato, Tomoyasu Nakano, and Masataka Goto. 2015. TextAlive: Integrated Design Environment for Kinetic Typography. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (Seoul, Republic of Korea) *(CHI '15)*. ACM, New York, NY, USA, 3403–3412. https://doi.org/10.1145/2702123.2702140

[20] Jun Kato, Takashi Nakashima, Hideki Takeoka, Kazunori Ogasawara, Kazuma Murao, Toshinari Shimokawa, and Masaaki Sugimoto. 2013. OpenPool: Community-based Prototyping of Digitally-augmented Billiard Table. In *Proceedings of the 2nd IEEE Global Conference on Consumer Electronics (IEEE GCCE '13)*. 175–176. https://doi.org/10.1109/GCCE.2013.6664790

[21] Jun Kato, Daisuke Sakamoto, and Takeo Igarashi. 2013. Picode: Inline Photos Representing Posture Data in Source Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Paris, France) *(CHI '13)*. 3097–3100. https://doi.org/10.1145/2470654.2466422

[22] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, Article 174, 11 pages.

[1] Programming as Communication | junkato.jp. https://junkato.jp/programming-as-communication/

https://doi.org/10.1145/3173574.3173748

[23] Janis Lena Meissner, John Vines, Janice McLaughlin, Thomas Nappey, Jekaterina Maksimova, and Peter Wright. 2017. Do-It-Yourself Empowerment as Experienced by Novice Makers with Disabilities. In *Proceedings of the 2017 Conference on Designing Interactive Systems* (Edinburgh, United Kingdom) *(DIS ’17)*. Association for Computing Machinery, New York, NY, USA, 1053–1065. https://doi.org/10.1145/3064663.3064674

[24] Brad A. Myers, John F. Pane, and Andy Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47, 9 (9 2004), 47–52. https://doi.org/10.1145/1015864.1015888

[25] Lexico powered by Oxford. [n.d.]. Environment | Meaning of Environment by Lexico. Website. Retrieved January 16, 2020 from https://www.lexico.com/definition/environment.

[26] Fernando Pérez and Brian E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007), 21–29. https://doi.org/10.1109/MCSE.2007.53 arXiv:https://aip.scitation.org/doi/pdf/10.1109/MCSE.2007.53

[27] Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. 2017. Codestrates: Literate Computing with Webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST ’17)*. Association for Computing Machinery, New York, NY, USA, 715–725. https://doi.org/10.1145/3126594.3126642

[28] Ben Shneiderman. 2002. Promoting Universal Usability with Multi-Layer Interface Design. *SIGCAPH Comput. Phys. Handicap.* 73–74 (6 2002), 1–8. https://doi.org/10.1145/960201.957206

[29] SideFX. [n.d.]. Houdini | 3D Procedural Software for Film, TV & Gamedev | SideFX. Website. Retrieved January 16, 2020 from https://www.sidefx.com/en/products/houdini.