Towards Blocks-based Prototyping of Web Applications

Robert Holwerda, Felienne Hermans
Dept. of Software and Computer Technology
Delft University of Technology
Delft, The Netherlands
{R.N.A.Holwerda, F.F.J.Hermans}@tudelft.nl

***Abstract*—The current generation of block languages, with its focus on teaching programming to novices, has not been designed for professionals. In this paper, we argue that blocks-based languages aimed at professional end-user developers face requirements that present challenges to the user interface design of such languages. We discuss three aspects that set potential professional use of block interfaces apart from educational use with children and students, and their implications for the design of blocks-based language editors. These aspects are that professionals: (1) require the editor to support high-productivity, (2) should not be limited by a simplified run-time environment, and (3) need the blocks editor to provide support for working with large programs. These three aspects provide research avenues for extending the usefulness of blocks-based language interfaces. We intend to explore these aspects with the design and development, of a blocks-based prototyping system for web designers. We report some preliminary results from an initial user experience-study in which 4th-year web design students were exposed to a blocks-based version of a language they already knew.**

I. INTRODUCTION

In [1], David Bau et al. ask why professionals do not program with block interfaces. They discuss drawbacks of block interfaces compared to text editors, highlighting higher viscosity (resistance to change), lower information density, search and navigation issues and suitability for source control systems. But many of the well-known blocks-based programming systems are focused on educating children and offer a simplified programming domain, e.g. 2D multimedia for Scratch [2], Greenfoot [3] and Pencil Code [4], 3D worlds for Alice [5], or Augmented Reality games for TaleBlazer [6]. Such simplified domains are not suitable for professionals, even if we understand the notion of professional to include not just trained software engineers, but also end-user programmers.

MIT App Inventor is richer, and its facilities support a large range of applications, which can make it attractive to some classes of professional users. The game development system Construct 2 is another example of a blocks-based system suitable for professional use. But for both these systems, we are not aware of published research into the aspects of their block interfaces that help or hinder professionals. With regard to the learnability of languages, it seems plausible to extrapolate the success that block languages have shown with students to professionals, but this extrapolation bears investigation. But professional use of block interfaces, even in the category of end-user programming, adds new avenues of research to the study of blocks, including:

Continual use: Someone who has incorporated a blocks-based tool into their professional life, may use the tool many times and desire a high-productivity while using the tool. He or she is not going to stop using the tool after an introductory course in programming has finished. The expectation of continual use may well be what justified the initial learning investment. The need for high productivity could be an important factor why the authors of [1] bring up viscosity, density, and navigability as drawbacks whose effect is felt stronger by professionals. And, judging by the amount of 'power-user' features in non-programmer application software like office-suites, design-tools like Adobe CC, and digital music production systems like Ableton Live, the desire for high-productivity features is not only found among trained software engineers.

Access to the underlying platform: The simplified programming domains used by educational block-languages, constitute a walled garden by blocking access to features of the underlying platform or important infrastructure, such as the network. To make, for example, learning HTML less difficult, the designers of the only blocks-based HTML editor (based on Droplet) deliberately reduced the number of HTML tags offered by the block interface from 109 to 40 [7]. Few HTML attributes are offered by the editor and no CSS features. People who edit HTML as part of their profession might very well find this too limiting. App Inventor, a relatively rich blocks-based programming system, also has limitations (e.g. database access, or widget layout), and its extensions facility is not available to block-language developers.

In an educational setting, this does not need to be a problem. "High ceilings" (few limits on what can be accomplished) can be less important than "wide walls" (support for wide range of explorations) [8]. But to a professional who is investing in learning a language, it may be important to know that the language will be usable not just for the coming project, but for many more projects afterward, whose requirements cannot be known yet. Modern programming platforms such as libraries, API's, frameworks etc., can be large. It is not known to what extent new UI designs can preserve the easy discoverability of blocks and their parameters, without boxing-in users looking to solve real-world problems.

One manifestation of this issue—that may be particularly interesting in the realm of block language design—is that, in some domains, multiple languages must be mixed to create a

typical solution. A web application prototype might use HTML, CSS, SQL, JavaScript, and Ruby, along with some DSLs for testing, configuration, and build tools. System operators might use shell scripts, Ansible playbooks, Python code and Docker files in a single project.

As blocks-based languages and text-based languages grow closer [1], research into blocks-based support for multiple languages could produce practical benefits to end-user programmers in such domains. Although many blocks-based languages have their own specific blocks editor, Blockly shows how a blocks-based editor can be designed to support multiple languages. We therefore use the term *block interface* to refer to the user interface design of the editor, with the understanding that a block interface can be shared by multiple blocks-based languages.

Larger scale programs: In a large data set of 250K Scratch programs [9], scraped from the Scratch community site in 2016, the average size of a program is 144 blocks. A large dataset of 15K spreadsheets recovered from the collapse of Enron, the average number of formula's in a spreadsheet is 2,223 [10]. Long-running projects by end-user programmers in a professional setting are more likely to grow over time than to shrink. A good user experience of managing the growing scale and complexity of block programs has not been a design-goal for the block languages listed in [1], with the exception of the upcoming GP language, whose design incorporates a modern module system [11].

Besides module-systems, there are more aspects to programming at scale with blocks. Two of the drawbacks mentioned in the first paragraph, lower information density and search/navigation issues, are related to the size of the program. And the third, viscosity, is discussed in [1] with respect to making small edits, but support for restructuring and refactoring will also decrease the resistance to change felt by the users. Many blocks-based systems have support for the rename refactoring, but the access that a blocks editor has to the abstract syntax tree provides opportunity for designing block interfaces that can support users making large scale changes to sizable programs.

II. Blocks-Based Prototyping for Web Designers

We will explore the design space of blocks-based language UI's for one class of professionals: interaction designers for web applications, who want to create prototypes of the interactivity in their designs. Interaction designers generally work in quick iterations, generating design alternatives and prototyping those to hone in on a final design [12]. But testing the behaviors of user interface elements requires software prototypes that can actually exhibit those behaviors in response to user action. In [13] the authors found that *"the behaviors that the designers wanted were quite complex and diverse, beyond what could plausibly be provided by a system that provided only a few built-in behaviors or a selection of predefined widgets, and therefore seemingly requires full programming capabilities"*, and the need to communicate with programmers made "*versions of the behavior much harder to iterate on*". While it is common for designers to have some rudimentary knowledge of programming, and web designers surveyed in [14] profess interest in technical aspects of web development, they also have no aspiration to become expert programmers.

This group of potential and current end-user programmers could benefit from blocks-based language(s) for the same reasons that students do: they facilitate learning programming concepts, make it easy to discover capabilities and features of blocks, and eliminate important classes of errors and frustration. But this group is also an audience of professionals, and the three aspects of professional use of block languages apply to them:

- Their iterative process and the need for live working prototypes of behavior make that an approachable and useful programming tool could see *continual use* by the designer. Note: with "continual use", we do not mean intensive day-to-day use. Sporadic, but recurring use would also, over time, add-up to many hours and cause a desire for high productivity.

- Given the complex and diverse behaviors [13] that need to be prototyped, designers cannot be boxed-in by a restricted set of blocks. They need *access to the underlying browser platform*: HTML, CSS and the API's exposed to JavaScript. This is an extensive platform: HTML5 has 109 elements and 114 attributes. CSS3 has 231 properties, excluding obsolete or experimental items. The Mozilla Developer Network lists 522 interfaces (excl. experimental or obsolete interfaces), each one containing multiple JavaScript functions and properties. The design of a blocks interface to browser capabilities will have to contend with much more blocks than educational blocks-based languages. Any attempt to select only those features that could be useful to interaction designers will have to be conservative. Also, there are rules regarding the composition of elements, properties and functions. Many blocks-based languages use particular shapes to convey rules about data types or syntactical categories, but this approach is unlikely to scale to the number of rules and restrictions in web languages. There are, however, some areas with room for serious simplification, because we focus on interaction designers, instead of e.g. back-end developers: The programming language, for example, does not need to be as complicated as modern-day JavaScript. Client-server communication and data persistence, both needed e.g. for prototyping social apps and requested by designers in [13], might also be simplified, perhaps along the lines of Netsblox's features [15] or the Firebase support in App Inventor.

- Not all prototypes for web interactions will be big, as some interactions or widgets can be tested in isolation. But to test the flow and interplay between multiple interactions, or to try complex behaviors, prototypes will grow to incorporate intricate logic, or to access advanced browser or internet API's. So, prototypes of non-trivial interaction designs become *larger scale programs*. Because these prototypes exist to facilitate iteration and experimentation with design alternatives, they exist to be changed. Improvements in support for large programs (information density, search/navigation, refactoring) is likely to have a measurable positive effect on the usefulness of the block interface.

III. Exploratory Study

We have conducted an exploratory user-experience study to get a first sense of what web-designers consider useful or problematic in using a blocks-based language. The participants were 10 students of a 4-year multimedia-design bachelor program, in their final year (9 men, 1 woman, ages: 20-26). On the basis of their 3,5 years of design-study, including a 5-month in-company internship, we consider these students to be sufficiently representative of our target audience of web designers. These students were also about 6 weeks into an entry-level programming course covering JavaScript and Arduino side-by-side. This gave them enough experience to perform programming tasks that are not absolute beginner's level, but not so much to make them averse to blocks-based languages just because of a long familiarity with text based code.

A. Study setup

The participants were observed while performing two programming tasks lasting about 70 minutes, using a Blockly variant, Ardublockly [16], aimed at creating Arduino software. The first programming task was identical to one they had already performed in C, as part of the coursework: implementing the logic and interaction for a slot machine, including modelling the rolling of the reels and winning and losing credits. The second task was also about slot machines, but participants were asked to reorganize and refactor an existing program that had lots of repeating code and badly named variables and functions.

Besides using an assignment the participants were familiar with, we chose a block language with semantics that was very close to a language they had used. This was done partly to prevent the cognitive load of solving the programming problem itself from dominating the user's perception of the experience of using a block interface, and partly to have the test involve a moderately large program (177 blocks to start with). We were, in effect, simulating a somewhat experienced professional performing a task with some complexity, but with a routine level of difficulty.

We would have preferred a blocks-based language oriented to web development, but could not find a block language that was close enough to JavaScript to support one of the more complex JavaScript tasks from the course. PencilCode was not used because it is designed to represent text-code as blocks, and therefore lacks several of the features of block interfaces that we were interested in, such as 2d program layout, drop-down menus in blocks, etc. ArduBlockly came close to supporting a familiar, moderately complex task for our participants. The fact that the tool was for another domain was not problematic because our participants had equal experience in both domains (JavaScript and Arduino). The two most important modifications to ArduBlockly were the addition of function parameters and variable declarations. We based our design of these on similar features in App Inventor, which is also a Blockly variant.

After performing the programming tasks, the participants were interviewed about their thoughts and feelings regarding the blocks interface. A screen recording of the task performance, including an eye-tracker overlay, was shown during the interview in order to prompt the participant to reflect on specific moments. This method, *retrospective think aloud*, is considered to yield data that sheds more light onto the cognitive aspects of the user experience than *concurrent think aloud*, where the user's comments tend more to describe actions and sensory perceptions [17].

B. Preliminary results

We are currently analyzing the results of this study, and cannot report definite findings yet. We will, however, informally discuss several things that were noted by the experimenter, based on the retrospective interviews, and on the questionnaires that all participants filled out later. The general response from participants to the idea of blocks-based programming was positive, and several inquired about blocks-based tools they could start using right away, for web development or game development.

Discoverability: The way the tool offered blocks in a toolbox panel, and the way each block displayed its features (parameters, drop-down menus etc.) was appreciated as a major benefit by the participants. This was not unexpected, but we note that these participants were already familiar with the commands and parameters used to program Arduinos. For most, however, the test-session came two or three weeks after they handed in their final Arduino-assignment, and about half the participants remarked that they had forgotten many names, options and other intricacies of the platform. Such gaps, of weeks or even months, may also be typical for many end-user programmers, because for them programming is not a day-to-day activity. So, the high discoverability afforded by most block interfaces remains an important feature, even for experienced end-user programmers.

Structural editing: All participants made remarks comparing blocks to text-based code, but there were only a few remarks to the effect that text was less restrictive or quicker to edit. Those few remarks were aimed at editing small arithmetical expressions, supporting the argument about making small edits in [1]. Otherwise, participants broadly expressed appreciation for the indivisibility and tangibility of the blocks and the way this prevented many syntax mistakes.

One aspect of the structure editor of Blockly that was commented on both positively and negatively, was the fact that some blocks can be reconfigured. To add an else-branch to an if-statement, or formal parameters to a procedure definition, the user opens a secondary block editor where he or she can change slots and labels of the block. Most participants had difficulty understanding this interface. Many participants, however, did appreciate that they could make such important changes without having to replace the block, and complained about situations where this was not possible. Other attempts to change blocks without replacing them were also observed. This desire to change existing blocks (instead of taking a structure apart in order to reassemble the structure using a new block) had, in reports of participants, to do both with speed of editing, and also with a worry that they might introduce errors while assembling the new structure. This worry grew with the number of blocks involved. From this, we conclude that support for refactorings at all levels could strengthen the appeal of structure editors like blocks, provided the UI is more intuitive than the one in Blockly.

Speed of manipulation: The drag-and-drop style of manipulation, triggered mixed feelings. It seemed to work well for initial exploration of possibilities, and strengthened the sense of tangibility, but a large majority of participants began looking for faster ways of manipulating once the block interface was understood. They asked about keyboard shortcuts and search facilities. This observation supports the idea that a GP-style of keyboard interface will be important, even for block interfaces that do not aim to help students transition to text-based editors. Many participants also tried, in vain, to use UI-features they knew from design-software (e.g. Photoshop, Sketch), like using modifier keys to select multiple blocks, or to cause the drag and drop interactions to copy the blocks. Rather than dismissing their requests for features found in graphical design software as mainly a desire to work in familiar ways, we surmise that these design-tools have been perfecting drag-and-drop direct manipulation interfaces for decades, and that blocks-based interfaces could learn to support high-productivity work styles by analyzing the UI-design of these professional design tools.

Overview issues: To simulate the experience of working with a moderately large program, the participants were given programs of 177 block in the first task, and 328 blocks in the second task. Most participants remarked that the 2D arrangement of sets of blocks was not helpful in either getting a quick overview of the code or in looking for a specific part of the program. For navigating and searching code, they preferred a sequential ordering of procedure definitions. For some, the situation improved when they got a sense of the thematic layout used in the programs, but then worsened when their own placement of new sets of blocks did not adhere to the same intuitive logic. On the other hand, we observed several participants using the natural scratch-space afforded by the 2d canvas to try out constructions before connecting the result to the actual program. If professional end-user developers benefit from the 2D canvas in this way, then new designs may be needed to help with scanning and searching the canvas.

Feedback: we also received requests for more (and better visible) feedback about problems while editing (e.g. type errors). Participants were trained in both C (compile time errors) and JavaScript (run-time errors). In response to the block interface, they started to want *edit-time* error messages, even though only one of the ten participants had any experience with such a feature in a modern IDE. In Scratch, no error-messages are ever given [2]; in Blockly, they are hidden behind a small icon. For professionals, a block interface that shows much feedback may be more appropriate to their goals of high productivity, and managing larger scale complexity.

The positive reactions to the high discoverability and structured editing suggest that blocks-based language UI's can work well for professional web designers. The problems encountered with changing blocks, editing speed, and overview call for enhanchments to the design of block interfaces to improve the usefulness for professionals. The issue of feedback indicates a design opportunity: blocks can be made to provide much more in-line feedback to the user. In addition to error messages, blocks could present results from static analysis (e.g. a type inferencer) or data from live execution (intermediate values, exceptions). For educational use with children, such inline feedback may overcomplicate the interface. For professional users, such inline feedback could increase productivity and help manage large scale programs, justifying the added complication of the UI.

IV. Conclusion

We have presented three considerations for the design of blocks-based language environments aimed at professional end-user programmers. Taken together with the preliminary results of our user-experience study with students on the brink of becoming professional web designers, we conclude that block interfaces do hold promise for this group, but research and design are needed to address these three considerations.

References

[1] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Learnable Programming: Blocks and Beyond," *Commun. ACM*, vol. 60, no. 6, pp. 72–80, May 2017.

[2] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch Programming Language and Environment," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, pp. 1–15, Nov. 2010.

[3] N. C. C. Brown, A. Altadmri, and M. Kolling, "Frame-based editing: Combining the best of blocks and text programming," in *Proceedings - 2016 International Conference on Learning and Teaching in Computing and Engineering, LaTiCE 2016*, 2016, pp. 47–53.

[4] D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens, "Pencil Code : Block Code for a Text World," in *14th International Conference on Interaction Design and Children*, 2015, pp. 445–448.

[5] S. Cooper, "The Design of Alice," *ACM Trans. Comput. Educ. Artic.*, vol. 10, no. 15, 2010.

[6] "TaleBlazer." [Online]. Available: http://www.taleblazer.org/. [Accessed: 17-Jul-2017].

[7] S. Aggarwal, D. A. Baau, and D. Bau, "A blocks-based editor for HTML code," in *Proceedings - 2015 IEEE Blocks and Beyond Workshop, Blocks and Beyond 2015*, 2015, pp. 83–85.

[8] M. Resnick and B. Silverman, "Some Reflections on Designing Construction Kits for Kids," in *Proceeding of the 2005 conference on Interaction design and children (IDC '05)*, 2005, pp. 117–122.

[9] E. Aivaloglou, F. Hermans, U. Rey, and J. Carlos, "A Dataset of Scratch Programs : Scraped , Shaped and Scored," in *2017 IEEE/ACM 39th International Conference on Software Engineering*, 2017, pp. 6–9.

[10] F. Hermans and E. Murphy-Hill, "Enron's Spreadsheets and Related Emails: A Dataset and Analysis," in *Proceedings - International Conference on Software Engineering*, 2015, vol. 2, pp. 7–16.

[11] Y. Ohshima, J. Monig, and J. Maloney, "A module system for a general-purpose blocks language," in *Proceedings - 2015 IEEE Blocks and Beyond Workshop, Blocks and Beyond 2015*, 2015, pp. 39–44.

[12] P. Campos and N. J. Nunes, "Practitioner tools and workstyles for user-interface design," *IEEE Softw.*, vol. 24, no. 1, pp. 73–80, 2007.

[13] B. Myers, S. Y. Park, Y. Nakano, G. Mueller, and A. Ko, "How designers design and program interactive behaviors," in *Proceedings - 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008*, 2008, pp. 177–184.

[14] B. Dorn and M. Guzdial, "Discovering Computing : Perspectives of Web Designers," *Icer*, pp. 23–29, 2010.

[15] B. Broll, A. Ledeczi, P. Volgyesi, J. Sallai, M. Maroti, and A. Carrillo, "A Visual Programming Environment for Learning Distributed Programming," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education - SIGCSE '17*, 2017.

[16] "Ardublockly." [Online]. Available: https://ardublockly.embeddedlog.com/. [Accessed: 17-Jul-2017].

[17] A. Hyrskykari, S. Ovaska, P. Majaranta, K.-J. Räihä, and M. Lehtinen, "Gaze path stimulation in retrospective think-aloud," *J. Eye Mov. Res.*, vol. 2, no. 4, 2008.