Calls of the Wild: Exploring Procedural Abstraction in App Inventor

Isabelle Li, Franklyn Turbak, Eni Mustafaraj
Computer Science Department, Wellesley College
Wellesley, Massachusetts, USA
Email: {ili, fturbak, emustafa}@wellesley.edu

***Abstract*—One of the most important computational concepts in any programming language is procedural abstraction. We investigate the use of procedures in MIT App Inventor, a web-based blocks programming environment for creating Android mobile apps. We explore how procedures are used "in the wild" by examining two datasets of App Inventor projects: all projects of ten thousand randomly chosen users and all projects of all prolific users (those users with 20 or more projects).**

Our data analysis indicates that procedural abstraction is a concept that is learned over time by some App Inventor users, but it is used relatively infrequently, and features like parameters and returning values are used even more rarely. Procedures are most frequently called only once, indicating that they are often used to organize code rather than to reuse it. Surprisingly, 10% of declared procedures are never called, suggesting that this situation should be flagged by the environment.

I. Introduction

Blocks programming languages are a popular way to lower barriers to programming for those with little or no programming experience as well as for casual programmers and even seasoned programmers using unfamiliar domain-specific languages [1]. For example, MIT App Inventor is a web-based blocks programming environment for democratizing the creation of apps for Android mobile devices [2] used by over 5 million people to create over 20 million app projects.[1]

The open-ended nature of blocks environments like App Inventor and Scratch (in which many projects are personally meaningful creations as opposed to more constrained programs specified as part of coordinated activities like courses) makes it challenging to investigate what users are learning when they create a sequence of projects. The long-term goal of our research is to use learning analytics on large datasets of projects to identify common conceptual difficulties experienced by App Inventor programmers and to improve the App Inventor programming environment and associated educational support materials to alleviate these difficulties [3].

One of the most important computational concepts in almost every programming language is *procedural abstraction*—the notion that a computational pattern can be captured in a (possibly parameterized) declaration (such as a procedure, function, or method) and can then be used by calling the declared entity on actual argument values that are used to fill the "holes" specified by the parameters. Procedural abstraction has numerous aspects. From a software engineering perspective, it allows code to be decomposed into reusable parts that can be written, tested, and debugged independently but called multiple times. Indeed, one App Inventor textbook [4] introduces procedures in the context of the *Don't Repeat Yourself (DRY)* mantra, a software engineering principle popularized in [5]. From a cognitive perspective, procedures break programs into smaller chunks that are easier to think about, so even procedures that are called only once can help to make programs more understandable. Procedures also establish an *abstraction barrier* that separates the high-level behavior of the procedures from its low-level implementation details, permitting clients to use procedures based on their contracts without knowledge of their implementations, allowing implementers to improve all calls by changing a single declaration, and supporting the notion of data abstraction [6].

In this paper, we study how App Inventor programmers use procedures by using a learning analytics approach based on two large datasets of App Inventor projects "in the wild": all projects of ten thousand randomly chosen users (whom we call "random users") and all projects of users with 20 or more projects (whom we call "prolific users").

This work makes several contributions to the study of App Inventor in particular and blocks programming in general:

1) We give insight into how App Inventor programmers use procedures, a key computational concept essential for understanding the skill level of users and how they learn over time.
2) We identify issues with procedure use that can be addressed by improvements in the App Inventor environment and educational materials.
3) The methodology we develop for studying procedures can be used to study other key computational concepts in App Inventor and other blocks languages.

II. App Inventor Projects and Procedures

We start with a concrete example: a `MyMoleMash` project that is a variant of a `MoleMash` tutorial program commonly used as an example of a simple game and a first program illustrating procedures (e.g., [4, Ch. 3]).

An App Inventor program is called a *project*. It can have multiple *screens*, each of which is specified independently. Our example (and most App Inventor projects) have a single screen. A screen consists of components that are chosen in a drag-and-drop *Designer* window (Figure 1). Components

[1] http://appinventor.mit.edu/ai2stats

978-1-5386-2480-7/17/$31.00 ©2017 IEEE

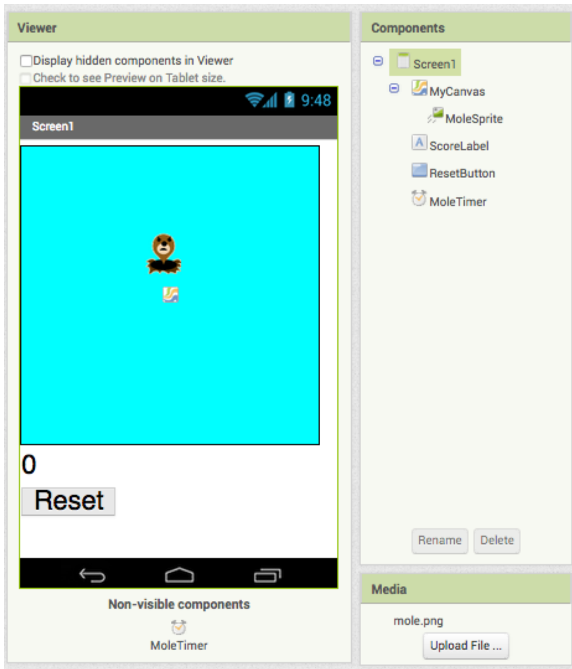


Fig. 1. Two panes of the Designer window for *MyMoleMash*.

include visible parts of the user interface. In `MyMoleMash`, these are: a sprite named `MoleSprite` that can be moved within the canvas `MyCanvas`; a text label named `ScoreLabel` initialized to 0; and a button named `ResetButon`. A screen can also have functional components that are not visible but contribute to app behavior. In `MyMoleMash`, the only functional component is the `MoleTimer`, which specifies an action that occurs every half second. There are dozens of other functional components that encapsulate mobile device features, including a camera, a sound player, a text-to-speech converter, a speech recognizer, and a GPS location sensor.

The behavior of a screen is specified in the *Blocks Editor*, in which visual blocks representing program syntax nodes are dragged out of *drawers* of related blocks onto a workspace, where they are clicked together to assemble programs (Figure 2). All program behavior is initiated by *event handlers* associated with components [7]. In `MyMoleMash`, there are three event handlers: `MoleTimer.Timer` moves the mole randomly every time it fires; when the mole is touched, `MoleSprite.Touched` increments the score and moves the mole to a random location; and `ResetButton.Clicked` resets the score to 0 when the button labeled `Reset` is clicked.

`MyMoleMash` has two procedure declarations. `MoveMole` is a parameterless procedure that moves `MoleSprite` to a random location on the canvas; it is called twice, in the `MoleTimer.Timer` and `MoleSprite.Touched` handlers. `DimensionRandom` is a procedure with two parameters (`canvasDimension` and `moleDimension`) that returns a random number between 0 and (`canvasDimension` − `moleDimension`); this is called twice in `MoveMole` to find random X and Y coords for `MoleSprite`. App Inventor procedure declarations can have any number of parameters, and the corresponding argument positions on the procedure call blocks are annotated with these parameter names.

App Inventor has two kinds of procedure declarations: (1) *fruitful* procedures, like `DimensionRandom`, that specify a return value through the `result` slot, and whose associated call blocks are value-denoting *expressions* that compose horizontally; and (2) *nonfruitful* procedures, like `MoveMole`, that do not return a value, and whose whose associated call blocks are action-performing *statements* that compose vertically.[2]

III. Related Work

The work most closely related to ours is an exploratory data analysis of 233K nonempty recently shared Scratch projects by Aivaloglou and Hermans [8]. They studied how frequently certain concepts and abstractions were used in these projects as evidenced by usage of particular blocks. They found that 7.70% of these projects use *custom blocks*, which are Scratch's way of specifying nonfruitful procedural abstractions. Most procedures (62.32%) are called exactly once, and 5.06% are never called. A majority of procedure definitions (55.57%) have no parameters, while 19.48% have only one. A notable difference between their work and ours is that they study only projects, while we study all projects of certain users.

Also closely related to our work is learning analytics for blocks programming that focuses on how block usage changes over time in user programs. For example, work on *skill progression* in Scratch [9], [10] explores the depth of projects (measured by total number of blocks used) vs. their breadth (measured by the number of different blocks types used). Another Scratch study introduces a notion of *learning trajectory* based on the changing vocabulary of blocks used in projects over time [11].

Adapting Brennan and Resnick's computational thinking framework [12] to App Inventor, Xie and Abelson develop a notion of *computational concept* that includes six concepts: procedure, variable, logic, loop, conditional, and list [13]. They also adapt the Scratch skill progression and learning trajectory work to investigate the breadth and depth of App Inventor projects over time, using a subset of the same prolific users dataset that we use in our work. While the blocks set they study includes fruitful and nonfruitful procedure declarations and calls, these are considered together with blocks for the other computational concepts. In contrast, our work does a deep dive into the analysis of procedure blocks only.

IV. Datasets

The App Inventor programming environment is provided as a web-based service at http://ai2.appinventor.mit.edu. To

[2]In some languages fruitful and nonfruitful procedural entities are distinguished by name (e.g., Pascal's nonfruitful "procedures" vs. fruitful "functions") or by return type (e.g., Java's nonfruitful `void` methods vs. non-`void` fruitful methods). In many languages, such as JavaScript, Python, Scheme, Standard ML, etc., a single term (such as "function" or "procedure") is used for both kinds of declarations, and nonfruitful declarations are distinguished from fruitful ones by either omitting an explicit `return` statement or returning a special or uninteresting "don't care" value (e.g., Python's `None` value or Standard ML's `unit` value).

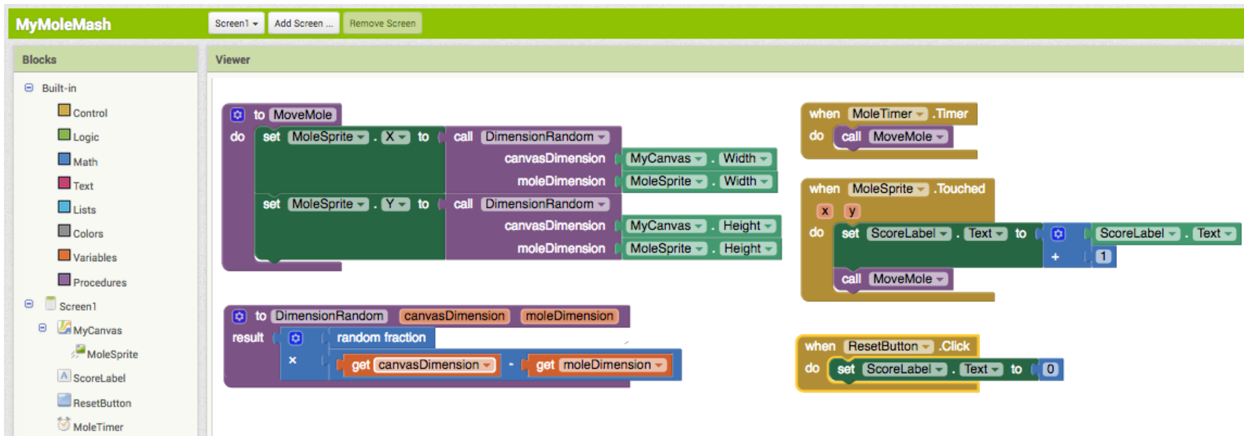


Fig. 2. The Blocks Editor for `MyMoleMash`. The program has three event handlers (`MoleTimer.Timer`, `MoleSprite.Touched`, and `ResetButton.Clicked`), the first two of which call the nonfruitful parameterless procedure `MoveMole`. The body of the `MoveMole` procedure declaration contains two calls to the fruitful `DimensionRandom` procedure, which has two parameters.

use this service, a person must supply credentials for a Gmail address, at which point they are considered to be "registered" App Inventor users with an App Inventor account. Any projects users create and modify are automatically stored in the cloud, associated with their account.

In this study, we analyze two datasets that were originally used in other App Inventor learning analytics work by Xie and his collaborators [13], [14]:

- **10K random user dataset**: All projects of ten thousand users randomly chosen out of all App Inventor users, as of May 5, 2015. There are 30,983 projects in this dataset.
- **46K prolific user dataset**: All projects of all App Inventor users with at least 20 projects as of March 15, 2016. There are 46,320 such users with a total of 1,546,056 projects in this dataset. 159 of these prolific users are also in the 10K random users.

Why two datasets? Because the random user dataset was relatively small, and we knew from a preliminary analysis that these users had a small average number of projects, we expected that these projects might not use many procedures. In contrast, we expected that the prolific users would include many students taking courses in which they would be likely to learn about procedures and use them in their projects.

All dataset projects were deidentified, in the sense that information about a user's Gmail address was removed, and each user was subsequently identified only by a user number (after the order of users was randomized). However, project names and all other project information was maintained.

Each project is represented by a collection of files that includes a pair of files for each screen: one that specifies the components of the project in JSON format, and another that represents the blocks of the project in an XML format determined by the Blockly framework. A project also includes its creation and last modification times. We wrote a Python program that summarized the component, block, and time information of each project as a JSON file, and used these summaries for our subsequent analyses.

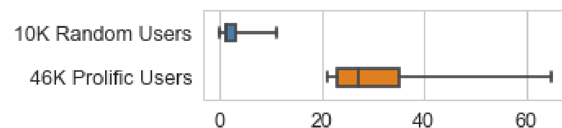


Fig. 3. Distribution of the number of projects for both the 10K random users and the 46K prolific users. The rightmost whisker represents the 95th percentile. The maximum number of projects is 226 for the random users and 634 for the prolific users.

To give an overall sense for these datasets, we present a few general statistics. Fig. 3 shows the distribution of the number of projects for users in the two datasets. For the 10K random users, the minimum number of projects is 0 (for users who visited the App Inventor site but never created a project), the median is 1, the average is 3.1, the 95th percentile is 11, and the maximum is 226. For the 46K prolific users, the minimum number of projects is 20, the median is 26, the average is 33.4, the 95th percentile is 65, and the maximum is 634.

The number of active blocks[3] per project is displayed in Fig. 4. The number of active blocks in a project is a crude metric for the complexity of the project. This idea is shown through our two datasets. The distribution of number of blocks per project is weighted more heavily to the lower numbers for the random users than the prolific users; indeed the median is 11 for the former and 23 for the latter. Interestingly, the maximum number of blocks for a prolific user project is 5,248, compared to 15,415 blocks by one of the random users. In fact, there are seven random users with at least one project that has a greater number of blocks than 5,248. This illustrates a general feature of the random user dataset: it tends to exhibit more variability than the prolific user dataset even though it has less than one quarter of the users and only about 2% of the number of projects as the prolific dataset.

[3] *Active blocks* excludes so-called *orphan blocks* that appear in the program but can't be executed, and so are effectively "commented out".

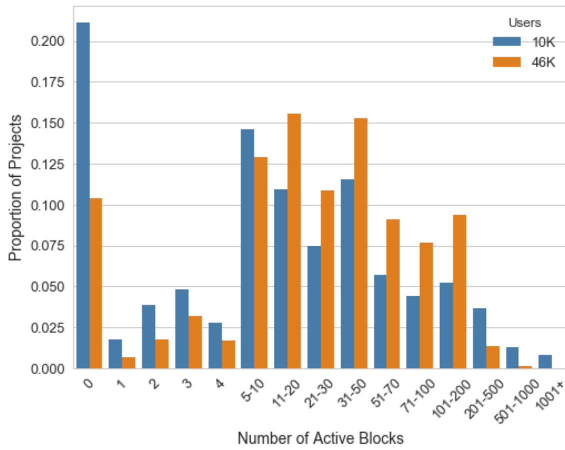


Fig. 4. The number of active blocks per project for both datasets.

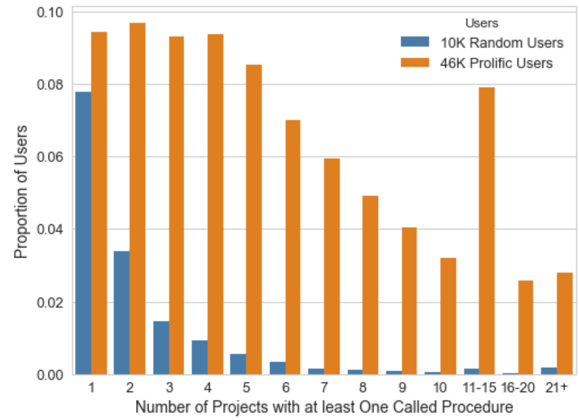


Fig. 6. Proportion of users having n projects with at least one called procedure.

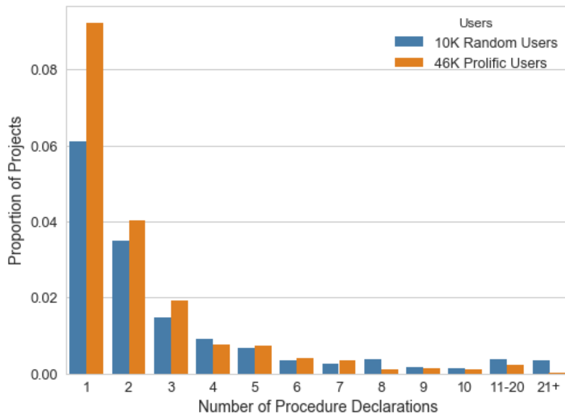


Fig. 5. Proportion of projects that contain n procedure declarations ($n > 0$). This chart excludes the 85% of random projects and 82% of prolific projects that contain no procedures.

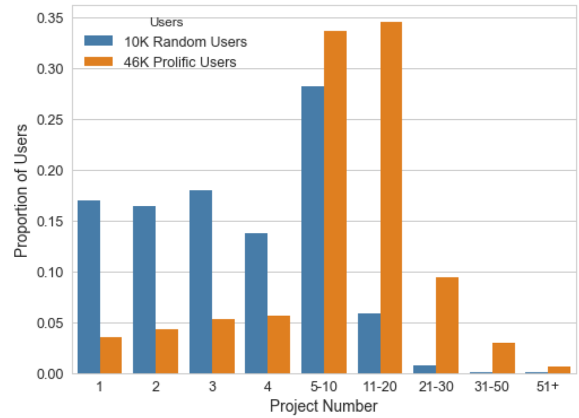


Fig. 7. Project number when users first use procedures

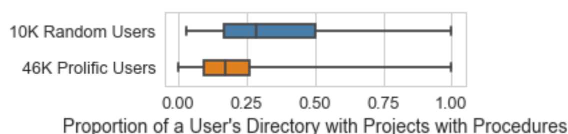
V. Analyses Involving Procedures

A. Procedure Declarations

Procedures aren't commonly used in App Inventor: 26,428 (85%) of random user projects and 1,267,643 (82%) of prolific user projects do not contain any procedure declarations. Fig. 5 shows the distribution of procedure declarations in the remaining projects. The percentage of projects with n procedure declarations decreases in a manner that is roughly $1/n$.

User statistics involving procedure declarations differ greatly between the two datasets. Only 1749 users (17.5%) of random users have some project in which a procedure is declared, but 39,873 (86.1%) of prolific users have such a project; this validates our expectation that the two datasets would differ in this regard. The large differences between the (minimum, median) numbers of projects for random users (0, 1) and for prolific users (20, 27) means that prolific users have many more opportunities to use procedures. Fig. 6 shows a breakdown of the proportion of users in each dataset that have n projects with at least one called procedure. As n increases, this proportion drops fast for random users, but for prolific users is flat as n ranges from 1 to 4, after which it drops in a linear fashion.

Another analysis that distinguishes the datasets can be seen in Fig. 7. This shows, for all users who eventually call a procedure in at least one of their projects, the project number (ordered by creation time) in which they first use a procedure. For the random users, 65% use a procedure early (within their first four projects), while this number is only 19% for prolific users. This suggests that there is a subset of the random users who come to App Inventor with prior programming experience, and so start using procedures in their projects right away. The much lower numbers for the prolific users suggest that either they come to App Inventor without knowing procedures (as might be the case for students taking a first programming course using App Inventor), or, even if they do have prior programming experience, they start their App Inventor experience with simple projects of the sort used in introductory tutorials (which do not include procedures). In a typical App Inventor course, a project similar to `MoleMash` (which uses zero-parameter nonfruitful procedures) would be introduced after simple apps, perhaps explaining the jump between projects 5 and 20 for prolific users.



(a) Distribution of users by the proportion of their projects with at least one called procedure This excludes users for whom the ratio is exactly 0.

(b) Distribution of users by the ratio of (1) the total number of called procedures they declared (in all projects) to (2) their total number of projects. This excludes users for whom the ratio is exactly 0. The rightmost whisker is the 95th percentile. The max random user ratio is 109 and the max prolific user ratio is 39.27 (omitted here).

Fig. 8. The user population for (a) and (b) is all users with at least one project in which there is at least one called procedure (1,522 random users and 39,226 prolific users).

Another procedure declaration analysis distinguishing the datasets is displayed in Fig. 8, which presents box-and-whisker plots that show two different distributions of users based on metrics that involve their use of procedures called at least once. In all plots, the population is the users that have some project containing a called procedure, i.e. users who never use a procedure are excluded. In Fig 8a, the metric is the proportion of a user's projects with at least one called procedure (necessarily between 0 and 1). In Fig 8b, the metric is the ratio of (1) the total number of called procedures they declared (in all projects) to (2) their total number of projects. In both metrics, the 10k random users show greater use of procedures, again suggesting that a greater fraction of users in this dataset have previous experience with procedures.

B. Procedure Calls

An analysis of procedures by the number of times they are called is shown in Fig. 9. The statistics for the two datasets are remarkably similar. The most frequent number of times a procedure is called is one (46% for random users and 44% for prolific users), suggesting that App Inventor procedures are often used to organize code into more easily understandable conceptual chunks than to capture reusable patterns and avoid code duplication. Although using procedures to enhance organization is important in textual languages, it is even more important in blocks languages because of the relatively large size of blocks compared to their textual analogs means that the density of code information is lower in blocks languages.

A surprising result is that about 10% of procedures in both datasets are never called. Why? We initially hypothesized that many users had dragged a procedure declaration block into the workspace without understanding its purpose and left it there. In this case, it would have an empty body. But uncalled procedures have empty bodies for only 18% of random users and 16% of prolific users. Fig. 10 shows the distribution of

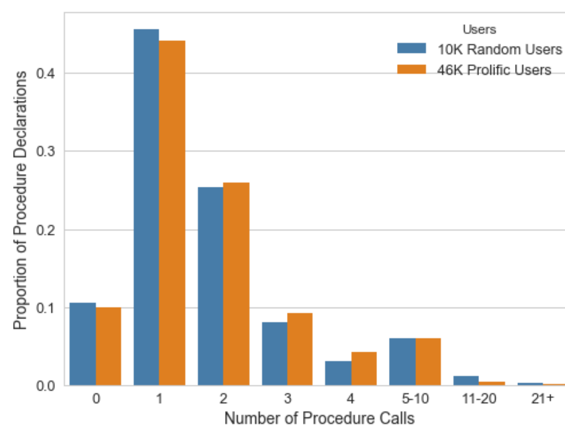


Fig. 9. Proportion of declared procedures that are called a given number of times.

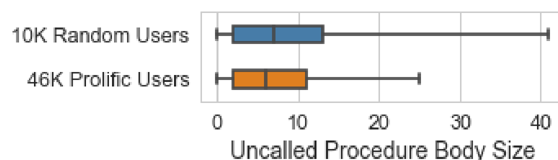


Fig. 10. Distribution of number of blocks within the bodies of uncalled procedure declarations. Rightmost whisker represents 95th percentile. Max procedure body size is 1136 for random users and 168 for prolific users.

body sizes for uncalled procedures, which cover a wide range. This suggests that other factors might be involved, such as:

- The project is incomplete. The procedure is under construction, was supplied as starter code, or was copied from another project, but has not yet been called.
- The procedure was called in a previous version of the project, but the user removed the calls because they were no longer necessary. They kept the procedure on the workspace in case they needed it again. This is equivalent to "commenting out" a might-be-used-in-the-future procedure declaration in text-based languages.
- The user does not understand that procedures need to be called in order to use them. The App Inventor interface does not help with this misconception; it does not show any procedure call blocks in the procedure drawer until after a procedure declaration block has been dragged onto the workspace.

Our procedure call results are similar to results reported in a study of 1.7M projects for 540K users of App Inventor Classic (an earlier version of App Inventor) [15]. That study found 6% of procedures were never called and that one was the most frequent number of calls for declared procedures.

C. Parameters

The distribution of parameters in procedure declarations is shown in Table I. Zero-parameter procedures are very common, accounting for 80% of procedures for random users and 71% for prolific users, swamping the numbers for one-

TABLE I
FREQUENCY OF PARAMETERS IN PROCEDURE DECLARATIONS.

Number of Parameters	10K Random Users	46K Prolific Users
0	80.0%	71.3%
1	15.1%	20.0%
2	2.6%	5.1%
3	1.7%	2.5%
4	0.5%	0.8%
5+	0.1%	0.3%

parameter procedures (15% for random, 20% for prolific) and all other cases (5% for random, 9% for prolific).

It might be that, in practice, zero-parameter procedures are what's needed to provide the appropriate level of abstraction. But we suspect that that there are other factors involved.

One reason that parameters may be used so infrequently is that the initial procedure block dragged from the procedures drawer has no parameters. Parameters can be added by clicking the gear icon in the upper left corner of the procedure block to reveal a mini blocks editor that allows adding, removing, and renaming parameters. But anecdotal experience suggests that many users don't know that the gear icon on a block allows them to change properties of that block.

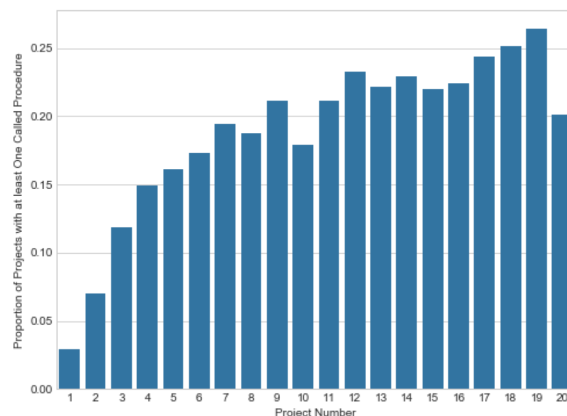
A second reason for infrequent parameters is that procedures in online App Inventor tutorials tend to be zero-parameter. For example, the `MoleMash` tutorial in [4, Ch. 3], which is used to introduce procedures, has two zero-parameter procedures. We modified it for this paper to include a two-parameter procedure. Of the 62 procedures that appear in the online tutorials at appinventor.mit.edu and appinventor.org, 40 have zero parameters, and these appear in the earlier tutorials.

D. Fruitful vs. Nonfruitful Procedures

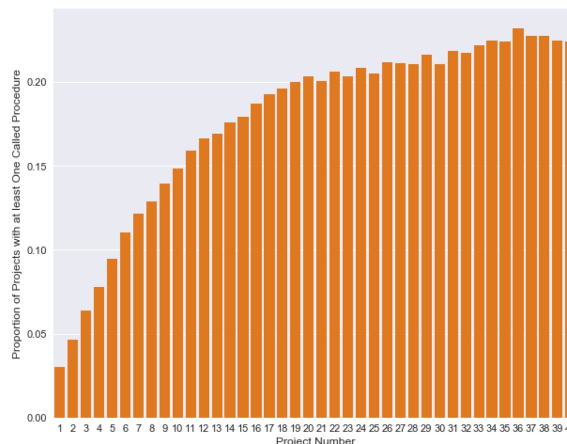
Table II indicates that in both datasets, nonfruitful procedure declarations and calls are much more common than fruitful ones. Again, one reason for this might be the lack of fruitful procedure examples in tutorials. Of 62 procedures that appear in the online tutorials at appinventor.mit.edu and appinventor.org, only 8 are fruitful, and these appear in 5 tutorials. Another reason may be that rather than returning values via a `return` statement (as is commonly done in imperative and object-oriented languages), fruitful procedures in App Inventor specify the return value via a `result` expression

TABLE II
STATISTICS ON FRUITFUL AND NONFRUITFUL PROCEDURE DECLARATIONS AND CALLS.

Type	Total	NonFruitful	Fruitful
10K Random Users Procedure Declarations	15,505	13,818 (89.1%)	1,687 (10.9%)
46K Prolific Users Procedure Declarations	638,305	554,007 (86.8%)	84,298 (13.2%)
10K Random Users Procedure Calls	32,259	27,940 (86.6%)	4,319 (13.4%)
46K Prolific Users Procedure Calls	1,205,737	1,067,630 (88.6%)	138,107 (11.4%)



(a) Results for 10K random users



(b) Results for 46K prolific users

Fig. 11. Skill progression in using procedures, as measured by the proportion of nth user projects (ordered by creation time) that contain at least one called procedure.

(as is done in functional languages). But when the fruitful procedure body involves features like local variables and loops, this can require using special-purpose blocks that allow the result value to "flow" to the `result` socket in the fruitful procedure declaration. We call this the *plumbing problem*, and have found that in practice it can be a barrier even to experienced programmers.

E. Skill Progression with Procedures

Fig. 11 shows the proportion of nth user projects (ordered by creation time) that contain at least one called procedure. Although prolific users necessarily have projects 1 through 20, not all users necessarily have an nth project for a given n, so Fig. 11 shows the proportion for all users that do have an nth project. Both datasets show a proportion of projects with procedures that fairly steadily grows from a small proportion for the first project to the range of 20 to 25% for later projects. There is more variability in the random user data because of the smaller number of projects having a larger project index. The growth over project index suggests that users are learning how to use procedures over time, but the fact that the growth

levels off at about a quarter of the projects is worrisome. It suggests that, after creating a substantial number of projects, users either aren't making projects complicated enough to require procedures, or they're failing to use procedures when they should be using them.

We have also made similar charts for skill progression involving (1) procedures with nonzero parameters that are called at least once and (2) fruitful procedures that are called at least once. Space does not permit the inclusion of these charts, but they are roughly similar in shape to those in Fig. 11 except for the final level approached (about 10% for procedues with nonzero parameters and 4 to 6% for fruitful procedures). These results bolster the conclusion that these two concepts are not learned well by App Inventor users.

VI. Discussion

A. Threats to Validity

Results about computational concepts from project datasets will be most meaningful when the projects are *original*, i.e., built from scratch by users based on their own ideas and current programming skills. However, it is likely that many projects in our datasets are *unoriginal*, i.e., they are created by following online tutorials, doing exercises in a class, or trying out or making minor modifications to existing projects shared by others (e.g., via App Inventor's *gallery* feature).

A previous study of App Inventor estimated at least 16.4% of projects were based on tutorials, as determined by project names [14]. It used a small tutorial set and didn't consider non-English versions of the project names, so this is most likely an underestimate. As part of this paper, we determined that 22% of procedure names in the random user dataset matched one of the procedure names used in tutorials at appinventor.mit.edu and appinventor.org, suggesting that many procedures in our datasets may be unoriginal, affecting our results.

Another source of unoriginality is that some users have many similar versions of a project, most likely created as checkpoints. App Inventor does not provide a mechanism for saving and restoring an earlier project version other than making a copy, nor did it have an undo capability until recently. So saving many versions of a large program is a strategy to avoid losing work. For example, we discovered that in Fig. 5, the bump at 8 procedures in the random user dataset was almost entirely due to one user who had 33 nearly identical versions of a project with 8 procedure declarations.

To understand what App Inventor users are learning and what misconceptions they have, we need to filter out unoriginal projects and focus on original ones. Our plan for doing this is sketched in [3]. It extends our previous work on determining project similarity by defining distance metrics between App Inventor projects represented as feature vectors [16].

B. Improving App Inventor

Our analyses so far suggest several ways to improve App Inventor with regard to procedures:

- Uncalled procedure declarations are surprisingly common, so they should be highlighted in some way. App Inventor currently puts various errors and warnings on problematic blocks, but uncalled procedure declarations currently carry no warning. They should, perhaps along with an easy way to create an associated call block.
- To highlight that procedures declarations need to have associated caller blocks, there should be additional ways to create caller blocks for a procedure other than opening the procedure drawer. For example, hovering over a procedure declaration with a mouse could open a menu option for creating a caller block in a way similar to hovering over a variable declaration gives a menu for creating getter and setter blocks for that variable [17].
- The procedure drawer could contain examples of declarations with at least one parameter. This would make it more obvious that App Inventor procedures have parameters for those not familiar with using the gear icon to edit the parameters of a procedure.
- When a user is copying a large block of code from an event handler or procedure to another, the App Inventor system might suggest that the user encapsulate that code into a procedure declaration, and could even automatically generate a candidate declaration.
- Software engineering approaches for automatically detecting opportunities for creating procedures to avoid code duplication [18], [19] could be adapted to App Inventor programs to allow an option for automatically refactoring the blocks on a screen by introducing (possibly parameterized) procedures and replacing the duplicated code by calls to these procedures.
- App Inventor would benefit from more tutorials involving procedures (especially fruitful procedures and procedures with parameters), as well as a help system that provides documentation/examples for blocks in context—e.g., how to declare and use procedure parameters and how to solve the plumbing problem for fruitful procedure bodies.

C. Classifying Users

The App Inventor environment does not collect demographic data about users, nor does it "know" the role in which people are using it. Some users come to App Inventor with significant prior programming experience, while many are programming newbies. Some users are students taking a semester-long class and will be engaged with App Inventor for months; others are casual programmers working on their own projects; yet others are just trying App Inventor out, perhaps in the context of an activity like Code.org's *Hour of Code*.

A user's skill level with procedures and other computational concepts can help to classify them. Other user data, such as their number of projects, the period in which they're engaged with App Inventor, and the overlap of the creation times of their projects with the project of others, can help to identify them as members of a coordinated activity, like a course or club [3]. Some prolific App Inventor users are teachers, which can often be deduced from the fact that they appear to have created numerous large projects around the same time when they upload their students' projects to grade them.

Automatically classifying users in terms of their role and expertise level could be used to customize the kinds of suggestions, examples, documentation, etc. that are offered to them in a more interactive version of App Inventor. Research in intelligent tutoring systems has long used user modeling to suggest activities for students based on their skill level [20]. More recent work has indicated that enhancing these models with student-specific parameters that take into account the speed of learning improves the predicting power of the models [21]. Furthermore, data-driven learning of student parameters can also reduce the need for embedding significant domain knowledge [22].

VII. Conclusion and Future Work

Our preliminary exploratory data analysis of procedures in App Inventor projects indicates that procedures are a concept that is learned over time, but they are used relatively infrequently, and features like parameters and returning values are used even more rarely. Procedures are most frequently called only once, indicating that they are often used to organize code rather than to reuse it. Surprisingly, 10% of declared procedures are never called, indicating conceptual confusions and suggesting that this situation should be flagged by the environment.

With regard to procedures, a next step is to use a feature-vector representation of projects (1) to filter out unoriginal procedures and repeat the analysis from this paper to see how this affects the results and (2) to approximate missed opportunities for proceduralization in a project.

We also plan to study other App Inventor features that support abstraction, such as lists, loops, and generic blocks. Preliminary investigations indicate these are also used rarely and have associated misconceptions. We hypothesize that the concrete nature of blocks may encourage a kind of "abstractionless programming" in which abstraction mechanisms will be rarely used unless they are somehow taught explicitly, possibly by interventions from the programming environment.

The similarity between our results and those in the exploratory analysis of Scratch [8] (e.g., few projects with declared procedures, most procedures are parameterless, most common number of times a procedure is called is one) suggests further in-depth investigations involving multiple blocks languages.

Finally, we imagine using the skill level exhibited with computational concepts like procedures to classify users and enable customized user feedback from the programming environment.

Acknowledgments

This work was supported by the Wellesley College Science Summer Research Program and an IBM Faculty Research Fund for Science and Math. The App Inventor datasets were provided by the MIT team's Jeff Schiller. Our analyses use a Python project summarization program that builds upon earlier work by Benji Xie and Maja Svanberg. Maja's work was supported by a Wellesley College Faculty Grants and by the National Science Foundation under grant DUE-1226216.

References

[1] D. Bau, J. Gray, C. Kelleher, J. S. Sheldon, and F. Turbak, "Learnable programming: Blocks and beyond," *Communications of the ACM*, 2017, to appear.

[2] D. Wolber, H. Abelson, and M. Friedman, "Democratizing computing with App Inventor," *GetMobile: Mobile Computing and Communications*, vol. 18, no. 4, pp. 53–58, Jan. 2015.

[3] F. Turbak, E. Mustafaraj, M. Svanberg, and M. Dawson, "Work in progress: Identifying and analyzing original projects in an open-ended blocks programming environment," in *Proceedings of the The 23rd International DMS Conference on Visual Languages and Sentient Systems (DMSVLSS 2017)*.

[4] D. Wolber, H. Abelson, E. Spertus, and L. Looney, *App Inventor 2: Create your own Android Apps*, 2nd ed. O'Reilly Media, Inc., 2014.

[5] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 2000.

[6] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs (2nd ed.)*. MIT Press, 1996.

[7] F. Turbak, M. Sherman, F. Martin, D. Wolber, and S. C. Pokress, "Events-first programming in App Inventor," *Journal of Computing Sciences in Colleges*, Apr. 2014.

[8] E. Aivaloglou and F. Hermans, "How kids code and how we know: An exploratory study on the Scratch repository," in *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*, 2016, pp. 53–61.

[9] C. Scaffidi and C. Chambers, "Skill progression demonstrated by users in the Scratch animation environment." *International Journal of Human-Computer Interaction*, vol. 28, pp. 383–398, 2012.

[10] J. N. Matias, S. Dasgupta, and B. M. Hill, "Skill progression in Scratch revisited," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*, 2016, pp. 1486–1490.

[11] S. Yang, C. Domeniconi, M. Revelle, M. Sweeney, B. U. Gelman, C. Beckley, and A. Johri, "Uncovering trajectories of informal learning in large online communities of creators," in *Proceedings of the Second (2015) ACM Conference on Learning @ Scale (L@S '15)*, 2015, pp. 131–140.

[12] K. Brennan and M. Resnick, "New frameworks for studying and assessing the development of computational thinking," in *Annual Meeting of the American Educational Research Association*, Vancouver, CA, 2012.

[13] B. Xie and H. Abelson, "Skill progression in MIT App Inventor," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2016, pp. 213–217.

[14] B. Xie, I. Shabir, and H. Abelson, "Measuring the usability and capability of App Inventor to create mobile applications," in *3rd International Workshop on Programming for Mobile and Touch*, 2015, pp. 1–8.

[15] J. Okerlund and F. Turbak, "A preliminary analysis of App Inventor blocks programs (showpiece/poster)," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*, Sep. 2013, abstract available at http://cs.wellesley.edu/~tinkerblocks/VLHCC13-abstract.pdf.

[16] E. Mustafaraj, F. A. Turbak, and M. Svanberg, "Identifying original projects in App Inventor," in *Proceedings of the Thirtieth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2017.*, pp. 567–573.

[17] F. Turbak, D. Wolber, and P. Medlock-Walton, "The design of naming features in App Inventor 2," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '14)*, Aug. 2014.

[18] R. Komondoor and S. Horwitz, "Eliminating duplication in source code via procedure extraction," Dept. of Computer Sciences, University of Wisconsin-Madison, Tech. Rep. 1461, 2002.

[19] T. J. Edler von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '14)*, 2014, pp. 85–94.

[20] A. T. Corbett and J. R. Anderson, "Knowledge tracing: Modeling the acquisition of procedural knowledge," *User modeling and user-adapted interaction*, vol. 4, no. 4, pp. 253–278, 1994.

[21] M. V. Yudelson, K. R. Koedinger, and G. J. Gordon, "Individualized Bayesian knowledge tracing models," in *International Conference on Artificial Intelligence in Education*. Springer, 2013, pp. 171–180.

[22] S. J. Lee, Y.-E. Liu, and Z. Popovic, "Learning individual behavior in an educational game: a data-driven approach," in *Educational Data Mining 2014*, 2014.