The Design of Naming Features in App Inventor 2

Franklyn Turbak
Computer Science Department
Wellesley College
Wellesley, Massachusetts, USA
Email: fturbak@wellesley.edu

David Wolber
Computer Science Department
University of San Francisco
San Francisco, California, USA
Email: wolber@usfca.edu

Paul Medlock-Walton
Scheller Teacher Education Program
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA
Email: paulmw@mit.edu

***Abstract*—Blocks languages, in which programs are constructed by connecting blocks resembling puzzle pieces, are increasingly used to introduce novices to programming. MIT App Inventor 2 has a blocks language for specifying the behavior of mobile apps. Its naming features (involving event and procedure parameters, global and local variables, and names for procedures, components, and component properties) were designed to address problems with names in other blocks languages, including its predecessor, MIT App Inventor Classic. We discuss the design of these features, and evaluate them with respect to cognitive dimensions and fundamental computer science naming concepts.**

I. Introduction

Blocks programming languages, in which coding is done by connecting drag-and-drop fragments shaped like puzzle pieces, are increasingly used in introductory programming environments. Over 15 million people have solved maze challenges in Blockly [1] as part of code.org's *Hour of Code* [2]. Millions more have used Scratch ([3], [4]) to create animations and games, MIT App Inventor [5] to build Android apps, and StarLogo [6] to experiment with multi-agent simulations.

Since 2009, we have taught and informally observed hundreds of App Inventor users (including both students and teachers from college, high school, and middle school) in the context of workshops, extracurricular activities, full-semester college courses, and online user forums. We observed that users of App Inventor Classic (AI1) had numerous difficulties with naming features, such as declaring and referencing global variables and parameters for events and procedures. We have designed and implemented improved naming features for App Inventor 2 (AI2) that are applicable to blocks languages in general. These features help blocks programmers declare variables and use them in the right scope, and also help them edit blocks code involving variables and other named entities, such as procedures, components, and component properties.

In this paper, we use the cognitive dimensions framework ([7], [8]) to present and evaluate the naming features of AI2. Our contribution is the design of naming features for blocks languages that respect fundamental naming concepts in computer science and mathematics (e.g., declaration vs. reference, scoping, consistent renaming) and address problems with naming in other blocks languages.

II. MIT App Inventor

MIT App Inventor is a blocks-based environment for creating Android mobile apps. An App Inventor project consists of a set of components and a program specifying their behavior. Components include visible user interface items (e.g., buttons, images, and text boxes) and non-visible items used in the app (e.g., camera, speech recognizer, GPS sensor). The program is written in a blocks language. For example, the AI2 blocks shown in Fig. 1 specify that a small circular dot should be drawn in Canvas1 wherever it is touched, and that a horizontal sequence of dots growing in diameter (vaguely resembling a comet) should be drawn in Canvas2 wherever it is touched.

In AI1, released in 2009, the blocks editor runs as a Java Web Start application. In AI2, released in Dec., 2013, the blocks editor runs in a web browser as a JavaScript program based on the Blockly blocks framework [1]. AI1 users with 1.4 million unique email addresses have made over 3.5 million projects, and AI2 users with nearly 0.5 million unique email addresses have made over 1.1 million projects.

III. Cognitive Dimensions Evaluation

Cognitive dimensions of notations [7] is a framework for evaluating notational systems like programming languages. Here we use the most relevant of the dimensions enumerated by Green and Petre for visual programming languages [8] to present the naming features of AI2 and compare them to those of other blocks languages, particularly AI1.

We were unaware of cognitive dimensions when we started this work, so it did not guide our designs. However, it is useful for explaining and evaluating the key aspects of our design, and it suggests how we can further improve AI2 naming features.

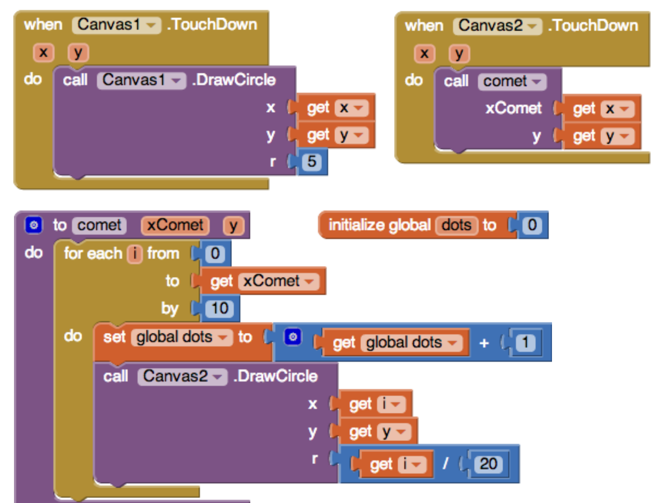


Fig. 1. Sample AI2 blocks program.

Error-proneness: This dimension measures how a notation leads users to make mistakes. This is the most important dimension for blocks languages, whose primary purpose is to reduce syntactic errors in text-based programming. Block shapes guarantee that blocks can be connected only in syntactically meaningful ways. The plug on the left of an AI2 expression block, which indicates it produces a value, only fits into sockets that consume values. Statements (with upper-left notches and lower-left nubs) compose in vertical sequences that fit into statement contexts, like those labeled `do` in Fig. 1. Labels on sockets help programmers remember the number and order of operands. Blocks are selected from *drawers* — menus of related blocks (e.g., math blocks, control blocks) — so it is not necessary to remember or type their names. Block nesting elucidates the tree-shaped structure of programs.

Naming operations include (1) declaring names (global/local variables, event/procedure parameters), (2) referencing declared names (to get/set their values) and (3) consistently renaming a declaration and all its references. Here we show how these are done in AI2 and argue that they reduce error-proneness relative to other blocks languages.

The main naming error is an *unbound variable*, which can come from forgetting to declare a variable, using a variable outside its scope of declaration, or misspelling a variable's name. Text languages provide little protection against unbound variables, but report them at compile time or run time. Modern integrated development environments like Eclipse use color-coding and/or icons to flag unbound variables. Blocks languages offer additional features to prevent or highlight them.

There are five key aspects to handling names in AI2 that help to reduce errors. First, all AI2 variable declarations (whether for event/procedure parameters, global/local variables, or loop indices) are represented as non-block entities: salmon-colored text boxes with rounded corners. E.g., Fig. 1 has event parameters `x` and `y`, procedure parameters `xComet` and `y`, loop index `i`, and global variable `dots`. Event handler and loop blocks have built-in name declarations. Procedure parameters are declared using Blockly's mutator mechanism, illustrated in Fig. 2. In contrast, AI1 parameter declarations are expressed with `name` declaration blocks that plug into parameter declaration sockets (Fig. 3). AI1's use of the same plug-and-socket metaphor for parameter declarations as for connecting value-producing expression plugs to value-consuming sockets is inherently confusing, since these two distinct classes of blocks can't be connected even though their shapes suggest they can. AI2 fixes this problem by making the visual representation of name declarations unrelated to plugs and sockets.

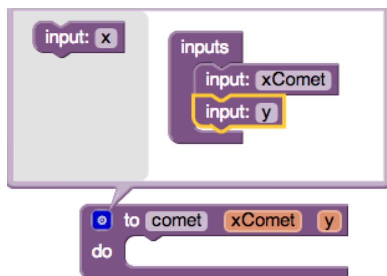
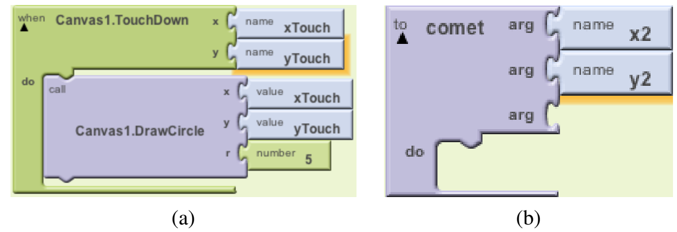


Fig. 2. Procedure parameters are added, removed, and reordered using a mini blocks editor called a *mutator*.

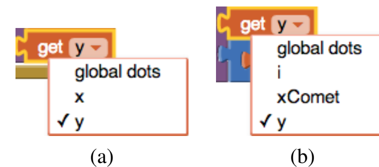


(a) (b)

Fig. 3. Event and procedure parameter declarations in AI1. Procedures always have one last empty *extensible socket* for adding another parameter.

Another problem with AI1 `name` blocks is that they can accidentally be unplugged. E.g., unplugging `name xTouch` in `Canvas1.TouchDown` in Fig. 3a leads to the cryptic runtime error message `call to 'Canvas1$TouchDown' has too many arguments (2; must be 1)`. This kind of error is surprisingly common. A study of 2300 AI1 users who generated runtime errors over a 3-week period found that over a quarter of the users experienced this kind of error [9]. AI2 eliminates these errors by getting rid of declaration blocks.

Second, all variable references in AI2 use drop-down menus that list only names that are valid in the current scope. AI2 uses variable getter/setter blocks with drop-down menus pioneered in TurtleBlocks/PictureBlocks [10]. Rather than taking the AI1 approach of listing `value` blocks (i.e., variable reference blocks) for all variables declared in the program in a My Definitions drawer, AI2 provides single generic get and set to blocks in its Variables drawer. As in TurtleBlocks/PictureBlocks, these blocks have a drop-down menu of names that initially displays no choice. When these blocks are dragged into the main workspace and connected to other blocks, the drop-down menu contains a list of only the names that are in scope at that point. For example, Fig. 4a shows the menu for the `get y` block in `Canvas2.TouchDown` in Fig. 1 and Fig. 4b shows the menu for the `get y` block in the `comet` procedure. The drop-down menu for a `set` block is exactly the same as the menu for a `get` block in the same context. Although the Blockly language has variable getter/setter blocks with drop-down menus, they list *all* variables and parameters in the entire program, just like the AI1 My Definitions drawer. This helps to prevent misspellings when names are manually typed, but does not help prevent out-of-scope variables.



(a) (b)

Fig. 4. Examples of `get` block drop-down menus listing in-scope variables.

Third, AI2 introduces a novel feature in which hovering with the mouse over a variable declaration displays a flyout menu with a `get` and `set` block instantiated for that variable (Fig. 5). One of these blocks can be selected from the flyout and dragged to the desired point of use. Once the getter/setter block is placed in context, the drop-down menu works as before and can be used to change the referenced variable. This addresses empirical problems with dragging getter/setter blocks from the Variables drawer: (1) novices often don't know to look there for these blocks and (2) the fact that the names in the blocks are blank until they are plugged into context does

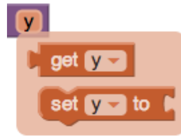
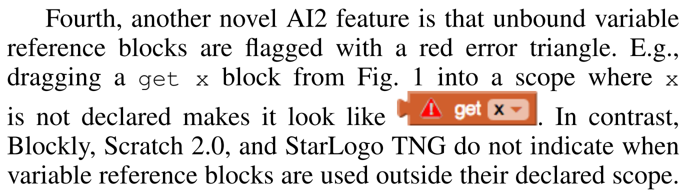


Fig. 5. Hovering over an AI2 variable declaration creates a flyout menu with `get` and `get` blocks for that variable.

not highlight their purpose. Informal observations suggest that selecting getters/setters from the flyout is much more popular than selecting them from the Variables drawer.

Fourth, another novel AI2 feature is that unbound variable reference blocks are flagged with a red error triangle. E.g., dragging a `get x` block from Fig. 1 into a scope where `x` is not declared makes it look like . In contrast, Blockly, Scratch 2.0, and StarLogo TNG do not indicate when variable reference blocks are used outside their declared scope.

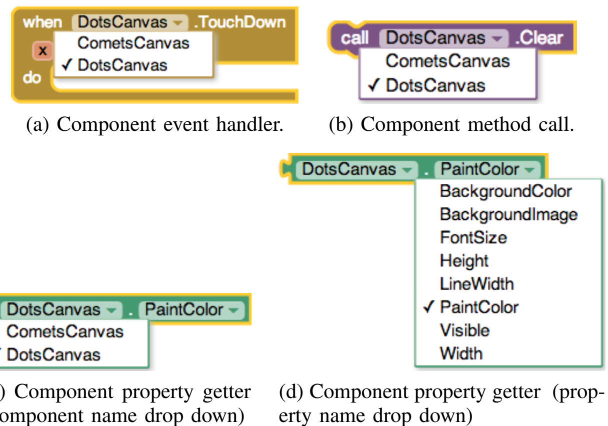
Fifth, changing the name of an AI2 variable declaration consistently renames all associated variable references in a way that prevents accidental name capture. E.g., in Fig. 1, changing the loop index variable `i` to `x` will automatically rename both `get i` blocks to `get x`. An attempt to rename `i` to `y` would accidentally capture the procedure parameter reference `get y` in `Canvas2.DrawCircle`, so this is not allowed; the automatically generated alternative `y2` is used instead. Blockly and Scratch 2.0 do not rename reference blocks when their declaration name is changed, which increases the likelihood of unbound variables. AI1 does consistently rename references when the declaration name is changed, but requires new names to be different than any other name used in the program. AI1's name uniqueness requirement violates a fundamental principle in computer science and mathematics that the choice of local names in constructs should be independent. E.g., using the local name x in the formula $\sum_{x=1}^{100} x$ should not preclude its use in some other formula; the scope of this particular occurrence of x is limited to the summation formula. AI2 provides the consistent renaming benefits of AI1 while respecting the CS principle of name locality.

Viscosity: This dimension measures how difficult it is to change a program. We have already discussed consistently changing the name of a declaration and all its references. Here we focus on other minor edits to code involving names.

Blocks programmers often want to change names in a block assembly. Suppose that the definition of an AI hypotenuse procedure involves creating the blocks in Fig. 6a. A subsequent step is to create the blocks in Fig. 6b. One way to do this is to drag a multiply block and two `value y` blocks from drawers and assemble them. Another way is to copy the assembly in Fig. 6a (both AI1 and AI2 allow copying/pasting arbitrary assemblies), remove the `value x` blocks, drag a `value y` block from the definitions drawer, copy it, and insert the two copies of `value y` into the copied multiply block. In either approach, many steps are required to achieve a simple goal.

Fig. 6. Two block assemblies from an AI1 hypotenuse procedure.

This task is easier in AI2. Starting with the assembly in Fig. 7a, the `x`-squared subassembly can be copied and placed in the second operand of the addition block (Fig. 7b). Then the getter blocks' drop-down menus can be used to change the two references of `x` to `y` (Fig. 7c). Such *in-place edits* made by choosing a different selection from a drop-down menu simplify block editing by avoiding the need to decompose blocks, make small changes, and reconnect them. AI2 also has drop-down menus on component blocks (Fig. 8) and procedure call blocks that reduce viscosity by permitting in-place editing In contrast, AI1 component and procedure call blocks have hardwired names. Changing these in a larger assembly requires the more cumbersome process of replacing the blocks by new blocks selected from drawers.



(a) Component event handler.
(b) Component method call.
(c) Component property getter (component name drop down)
(d) Component property getter (property name drop down)

Fig. 8. Drop-down menus in various AI2 component blocks.

Finally, editing AI2 block assemblies with variable references is aided by the fact that an unbound reference block remembers its original name, even though this name is not in the drop-down menu. When such a block is reinserted into the correct scope, the name stays the same, but the error triangle disappears (see Fig. 9). This is an improvement over TurtleBlocks/PictureBlocks, in which a disconnected getter's name is replaced with the special symbol `???`, and the old name must be reselected from the drop-down when the getter block is reconnected to the original scope.

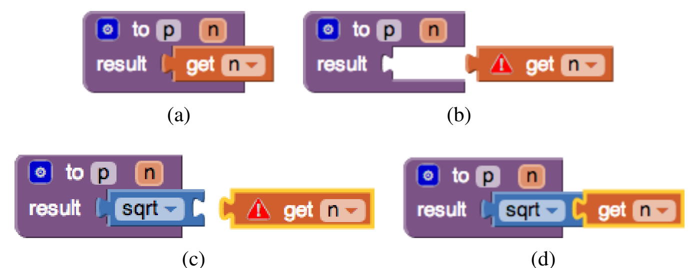


Fig. 9. Steps to change `p` from an identity to a square-root procedure.

Premature Commitment: This dimension measures ways in which the system requires doing things in a particular order. Like all blocks languages we know, AI1 and AI2 require that all named entities (variables, procedures, components) be declared before blocks that mention these names can be manipulated. This requirement interferes with tasks like assembling blocks for mathematical formulas, which cannot be done until the names mentioned by the formulas have been

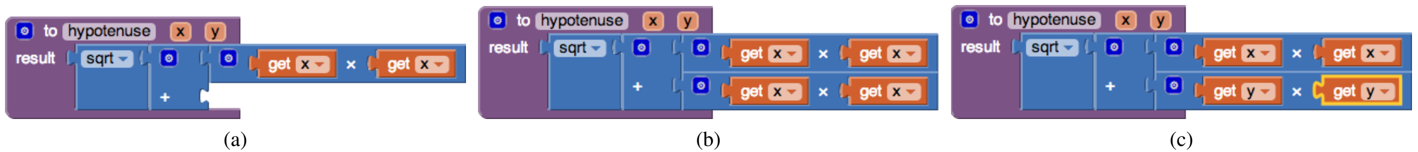


Fig. 7. Completing an AI2 hypotenuse procedure via copying and in-place edits.

declared. In contrast, the free-form nature of textual languages allows writing formulas referencing variables before wrapping them in a context that declares those variables.

Diffuseness: This dimension measures the space taken by notations. Blocks are typically much bigger than their textual counterparts, so less of the program can be viewed at once. AI2 getter/setter blocks have sizes similar to corresponding AI1 blocks except for global variable getters, where AI2 uses two words (`get global`) in place of the one word (`global`) in AI1. Parameter declarations in AI2 are much more concise, but global variable declarations are wider because the short keyword `def` has been replaced by `initialize global`.

Consistency: This dimension measures the interplay between similarity of notation and similarity of concepts. AI2 is more consistent than AI1 in its plug/socket notation because (1) it removes AI1's special cases for name declaration blocks that plug into parameter sockets and (2) it removes AI1's extensible sockets, as in procedure definitions (Fig. 3b), so that every empty AI2 socket indicates a hole that must be filled in order for the program to be complete.

AI2's representation of variable declarations and references is also more consistent than AI1's. In AI2, the same text box representation is used for all variable and parameter declarations. Additionally, all AI2 blocks that manipulate variables use the same salmon color scheme. In contrast, AI1 uses light blue for parameter declarations and all variable references (which adds to the confusion between `name` and `value` blocks) and a darker blue for global variable declarations and setters.

Hidden Dependencies: This dimension measures the visibility of important links between entities. No blocks language we know of (including AI1 and AI2) does a good job of directly showing (1) the declaration associated with the reference to a variable or (2) all references for a variable declaration. In AI1 and AI2, some sense of the second property can be gained indirectly by changing the declared name and seeing which getter/setter names change, but this is not very effective. Visualizing these dependencies is an area for future work.

IV. Conclusion and Future Work

We have developed naming features in AI2 that address numerous naming problems in AI1 and other blocks languages. Our design is a novel approach to naming in blocks languages that expresses fundamental naming ideas from computer science and mathematics, such as the relationship between name declarations and references, the locality and scope of names, and the consistent renaming of a declaration and all its references in a way that avoids variable capture. Our design improves upon the handling of naming in several cognitive dimensions: it is less error-prone, less viscous, and more consistent than in other blocks languages.

We are exploring two ways to improve AI2's naming features. First, to address the hidden dependency problem, we are investigating ways to visualize variable scope and the relationship between variable references and declarations. When dragging a getter/setter block, an explicit connection between the reference block and its declaration can be drawn and a shaded region of the declaration's scope can be shown to indicate all areas of the code in which its references can be used. Second, we are developing a feature that allows AI2 blocks to be represented as text within generic code blocks, and also allows these text-based blocks to be converted back and forth between regular AI2 blocks. This will allow writing code before declaring the names used in that code, addressing the premature commitment problem.

We plan to conduct a series of user studies that will help us guide the design of these improvements and measure the effectiveness of existing AI2 naming features.

Using cognitive dimensions to analyze naming features in App Inventor was a valuable exercise. It would be worthwhile to use cognitive dimensions to evaluate and compare existing blocks languages for a broader set of features.

Acknowledgments

This work was supported by Wellesley College Faculty Grants, by sabbatical funding from Wellesley College and the University of San Francisco, and by the National Science Foundation under grants DUE-1226216 and DUE-1225745.

References

[1] Neil Fraser, Blockly website, https://code.google.com/p/blockly, accessed May 16, 2014.

[2] Code.org, Hour of Code website, http://code.org/learn, accessed May 16, 2014.

[3] Scratch project, MIT Lifelong Kindergarten Group, http://scratch.mit.edu/, accessed May 16, 2014.

[4] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch programming language and environment," *ACM Transactions on Computing Education*, vol. 10, no. 4, Nov. 2010.

[5] App Inventor website, http://appinventor.mit.edu, accessed May 16, 2014.

[6] StarLogo TNG project, MIT Scheller Teacher Education Program, http://education.mit.edu/projects/starlogo-tng, accessed May 16, 2014.

[7] T. R. G. Green, "Cognitive dimensions of notations," in *People and Computers V*, A. Sutcliffe and L. Macaulay, Eds. Cambridge, UK: Cambridge University Press, 1989, pp. 443–460.

[8] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages and Computing*, vol. 7, pp. 131–174, 1996.

[9] Johanna Okerlund, Understanding App Inventor Runtime Errors, Wellesley Science Center Summer Research poster, Aug. 2013. Available at http://www.tinkerblocks.org/pubs.

[10] F. Turbak, S. Sandu, O. Kotsopoulous, E. Erdman, E. Davis, and K. Chadha, "Blocks languages for creating tangible artifacts," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '12)*, Oct. 2012, pp. 137–144.