Enhancing Abstraction in App Inventor with Generic Event Handlers

Evan W. Patton
MIT App Inventor
MIT
Cambridge, MA, USA
ewpatton@mit.edu

Audrey Seo
Computer Science Department
Wellesley College
Wellesley, MA, USA
aseo@wellesley.edu

Franklyn Turbak
Computer Science Department
Wellesley College
Wellesley, MA, USA
fturbak@wellesley.edu

***Abstract*—Work on code smells (undesirable programming patterns) in blocks languages has found that programmers often duplicate blocks code rather than abstracting over common patterns of computation using procedure-like features. For example, previous analyses of over a million MIT App Inventor projects have revealed that procedures are used surprisingly rarely in the wild and that many users miss opportunities for using procedural abstraction to avoid code duplication in their projects.**

In this work, we use data analysis to explain how particular features of App Inventor create barriers to abstracting over event handlers. In many cases, duplicated code in event handlers cannot be extracted into a procedure without using so-called generic blocks that abstract over a particular component (e.g., a label). Generic blocks are rarely used in practice, possibly because programmers do not know about them or find them difficult to use. But even proceduralization with generic blocks does not remove the need for duplicating the event handlers themselves.

We address these issues with two enhancements to App Inventor. First, we add generic event handlers, a new form of abstraction that allows specifying a single handler for all components of a particular type. Second, we add a way to easily convert between specific and generic blocks to facilitate *genericization*, that is, abstracting actions over a particular component to apply to a group of components of that type.

We also discuss related design choices and ways to encourage programmers to use the new features to avoid code duplication. Our work is an example of data-informed programming language design, in which the creation or modification of features is informed by the analysis of large datasets of programs from the language's target audience.

***Index Terms*—blocks programming, abstraction, procedures, code duplication, code smells, refactoring**

I. Introduction

Abstraction — the notion that systems can be understood and organized in ways that emphasize high-level processes and structures and suppress inessential information — is a key big idea in computer science. One form is procedural abstraction, where common computational patterns are captured via parameterized entities like procedures, functions, and methods whose behavior can be described by high-level specifications that hide the low-level implementation details. Computation can be expressed by invoking these entities on appropriate arguments rather than by duplicating code, making programs easier to read, write, debug, and modify.

Previous research suggest that procedural abstraction is underutilized in blocks programming languages. Studies of Scratch programs find that code duplication is a common code smell (i.e., undesirable programming pattern) [1], [2]. One study [3] of 1.5M App Inventor projects made by 46K prolific users (those with 20 or more projects) revealed that procedures are used surprisingly rarely in the wild. A followup study on the same dataset [4] found that over 86% of prolific users missed an opportunity for using procedures to avoid code duplication on certain patterns involving at least five blocks.

Code duplication is considered undesirable in most programming contexts. Indeed, it is listed as the first "bad smell" in Fowler's *Refactoring* book that introduced the term [5]. It can occur anywhere, but one study of student code quality in Java suggested that it is much less common in Java than in blocks languages [6]. Why might duplication be so common in blocks languages? The authors of [6] hypothesize that one reason is that these languages are targeted at a younger audience. Another reason may be that these languages are often used outside of traditional instructional contexts by people with little or no programming background, so they may not have been introduced to procedural abstraction or may not appreciate its benefits. The lack of object-oriented features in many popular blocks languages might affect aspects of code duplication relative to Java. Additionally, blocks programming environments make it very easy to copy/paste conceptual chunks of code and edit the copies, and there may be more (real or perceived) work to create and call procedural entities, at least for a small number of calls.

In this work, we focus on a different reason: the language and its environment may have specific barriers to abstracting over particular patterns. For example, Scratch's custom blocks cannot specify outputs and thus do not support procedural abstractions that return values. In App Inventor, we have discovered that capturing common code patterns in the bodies of event handlers often requires using so-called generic blocks that abstract over a component type (e.g., Label). Generic blocks are rarely used in practice, possibly because programmers don't know about them or find them difficult to use. But even using procedures with generic blocks does not remove the need for duplicating the event handlers themselves.

In this paper, we make several contributions to lower these barriers. First, we show that event handler duplication in App Inventor is common in practice and often involves generic

978-1-7281-4849-6/19/$31.00 ©2019 IEEE

blocks by analyzing over 1.5M user projects.[1] Second, we extend App Inventor with **generic event handlers**, a new form of abstraction that allows specifying a single handler for all components of a particular type, and describe issues encountered when using this feature. Third, we add a way to easily convert between specific and generic blocks to facilitate **genericization**, that is, abstracting actions on a particular component to apply to a group of components of that type. Finally, we describe other language design issues that still need to be resolved in this context and how users can be encouraged to use our extensions to avoid event handler duplication.

Our work is an example of what we call **data-informed programming language design**, in which language constructs are chosen or modified based on evidence from large datasets of programs from users targeted by the language.

II. App Inventor and Project Datasets

MIT App Inventor is a browser-based programming environment that democratizes the creation of mobile apps for Android devices [7]. For each screen of an app, a user first creates the user interface by dragging and dropping visual components (e.g., buttons) and behavioral components (e.g., a camera). Then the user specifies the behavior of these components by assembling blocks for these components in a blocks programming editor. In addition to supporting component-specific behaviors, the App Inventor blocks language supports traditional programming constructs like global and local variables, conditionals, loops, and procedures.

The App Inventor programming environment is provided as a free web-based service at http://ai2.appinventor.mit.edu. Any projects users create and modify are automatically stored in the cloud, associated with their Google account.

Our work involves analyzing two datasets of App Inventor projects used in several previous studies [3], [4], [8], [9]:

- **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.

These datasets, provided to us by the MIT App Inventor development team, were collected from the project cloud store. All dataset projects were deidentified, in the sense that information about a user's email 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 consists of a collection of files that includes for each screen a JSON file specifying the components and a Blockly XML file representing the blocks program. For this study, the second and third authors wrote a Python program that converted each project to a single JSON file combining all component and block information for all screens. This file format, which we call *JAIL* (*JSON App Inventor Language*), facilitates analysis of the abstract syntax trees for the blocks programs. Conversion to JAIL format failed for a small number of projects, so we ended up analyzing 30,851 of the 10K users, and 1,545,284 of the 46K users.

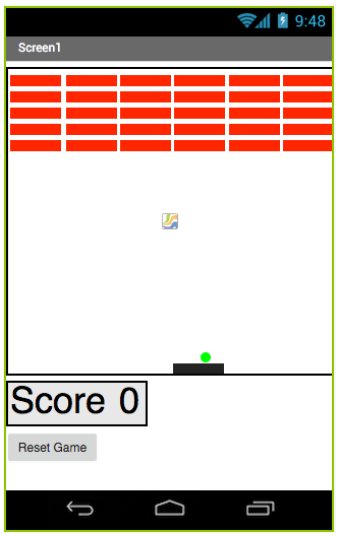


Fig. 1. The user interface for a brick breaker game for one of the prolific users. It has 30 stationary red bricks, a black paddle, a green ball, a score label, and a reset button. The ball bounces off the paddle, bricks and side walls of the canvas, but a brick also disappears whenever a ball bounces off it. The game ends when the ball hits the bottom wall of the canvas.

Some results of an earlier study on App Inventor procedures [3] are relevant here:

- Only 15% of the random user projects and 18% of the prolific user projects contain at least one procedure declaration. Whereas only 17.5% of random users have some project in which a procedure is declared, 86.1% of prolific users have such a project, indicating that prolific users are more likely to use procedures.
- About 10% of declared procedures in both datasets are never called, suggesting confusion on how to use them.
- 46% of declared procedures of random users and 44% of those of prolific users are called only once, indicating that they are often used for the purpose of organizing code rather than to eliminate code duplication.
- 80% of procedures for random users and 71% of procedures for prolific users have zero parameters, bolstering the impression that they are not being used to capture nontrivial computational patterns.

III. Event Handler and Procedure Example

The program for an App Inventor screen is expressed as a collection of event handlers along with procedure and global variable declarations [10]. To illustrate issues with event handler duplication, procedures, and generic blocks in App Inventor, we present key aspects of a brick-breaker game from the prolific dataset. Fig. 1 shows the user interface for this

[1]Some results (but not the algorithm) for a similar proceduralization analysis were previously presented in a 2-page extended abstract for a talk at another workshop that had no proceedings [4]. In Sec. IV, we describe in detail the algorithm and results for a slightly different analysis.

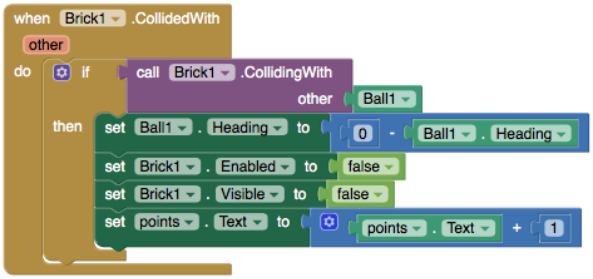


Fig. 2. A collision handler for `Brick1` that specifies what happens when it is hit by the ball. There are 30 copies of this handler (one for each red brick) in the project, so the project has significant code duplication.

game, which has 30 red bricks. Fig. 2 shows the collision handler for an `ImageSprite` component named `Brick1`, There are no procedures in this project; this handler is copied for each of the 30 bricks, causing significant code duplication.

Fig. 3 shows an alternative way to express the collision handler for `Brick1` that uses so-called *generic blocks* for `Brick1`'s `CollidedWith` method call and its `Enabled` and `Visible` property setters. In contrast with the *specific blocks* for the method call and property getters used in Fig. 2 (where a specific component is hardwired into the block), the generic blocks allow the component to be specified as a value. Users are unlikely to create the version of the handler in Fig. 3, but it is an intermediate step that explains generic blocks, motivates the proceduralization in Fig. 4, and reappears in the later discussion of specific/generic conversion in Sec. VI.

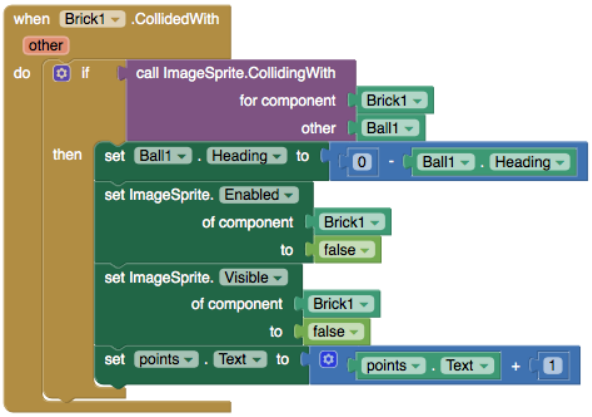


Fig. 3. An alternative collision handler for `Brick1` that uses generic blocks for `Brick1`'s `CollidedWith` method call and its `Enabled` and `Visible` property setters.

Fig. 4 shows how a `handleBallCollisionWith` procedure can abstract over the pattern of the `CollidedWith` handler for all 30 bricks. Unfortunately, in the version of App Inventor before the work reported here, there was no way to abstract over the same kind of handler for different components of the same type. So it would still be necessary to have 30 copies of the `CollidedWith` handler, one for each brick, even though the body of each is much simpler than before (consisting of a single call block for the `handleBallCollisionWith` procedure with the correct component block). This example shows that procedures could eliminate much, but not all, of the code duplication in the previous version of App Inventor (before our enhancements).

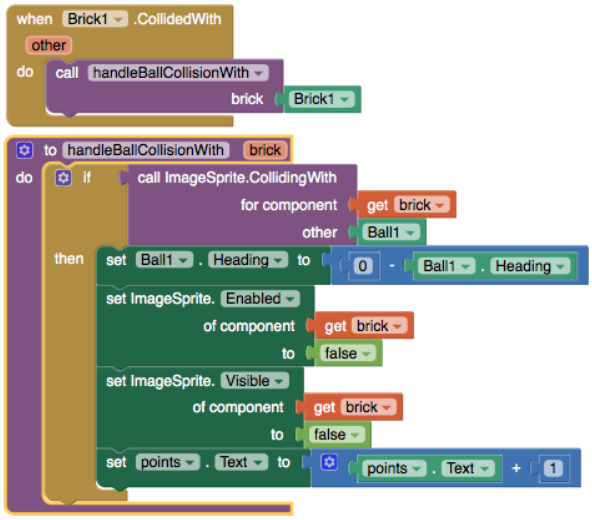


Fig. 4. Yet another alternative collision handler for `Brick1` that calls a `handleBallCollisionWith` procedure, passing a `Brick1` component block as the actual argument value for the `brick` parameter of the procedure. In the version of App Inventor before our work, the single `handleBallCollisionWith` procedure could be called for all 30 bricks, but there would still need to be 30 distinct `CollidedWith` handlers because there was no way to abstract over the component for a handler. So procedures could remove much of the code duplication, but not all of it.

IV. Event Handler Duplication Study

How commonly do nearly duplicate copies of event handlers (as in Sec. III) occur in practice? To answer this question, the second and third authors developed an algorithm to find nearly duplicate event handlers. Originally, the second author designed an algorithm to determine which event handlers have bodies that can be replaced by calls to an appropriate procedure [4]. But with the advent of generic event handlers (see Sec. V), the goal is now to find which event handlers can be replaced by an appropriate generic event handler.

A. Event Handler Equivalence Algorithm

The key idea of the algorithm is to specify an equivalence relation on event handlers such that two handlers are equivalent if there is a generic event handler that abstracts over them. Recall that an *equivalence relation* is one that is reflexive, symmetric, and transitive, and it partitions the elements of a domain into *equivalence classes*. Given all the event handlers in one screen of a project, we wish to partition them into equivalence classes such that the behavior of all the handlers in each class with more than one handler can be described by one generic event handler.

The equivalence relation works on the abstract syntax trees (ASTs) of the event handlers. Each block corresponds to an AST node. If a block B has sockets on the right, blocks plugged into those sockets are *expression nodes* that are children of B's node (ordered from top to bottom). Blocks that compose vertically are *statement nodes*, and a maximal length vertical sequence of blocks can be viewed as a single sequence statement node whose n children are the n individual nodes of the sequence (again ordered from top to bottom). Statement nodes serve as children in a few blocks, such as the branches of conditionals and the bodies of event handlers and

procedures. Event handlers, procedure declarations, and global variable declarations are top-level nodes that are the roots of distinct ASTs.

For simplicity, two event handler ASTs are considered equivalent if and only if they have *exactly* the same nodes-with-children tree structure and the corresponding nodes have the same *node type*, which is defined as follows:

- Literal leaf nodes have node types like `boolean`, `number`, `string`, and `component block`. E.g., since `17` and `42` both have node type `number`, they are considered equivalent even though the values are different.
- Getter blocks for global variables all have node type `getGlobal` regardless of the variable name. Similarly setter blocks for global variables have node type `setGlobal`. (See Fig. 10 for an example of a global variable.) So references to global variables with different names are equivalent. This allows abstracting over event handlers that reference different globals in the same AST position, a pattern that occurs commonly in practice but requires special handling when abstracting.
- In contrast, getters and setters for local variables have the variable names baked into their node type. For example, the getter for the `brick` variable in Fig. 4 has node type `getLocal_brick`. So references to local variables with different names are *not* equivalent.
- Perhaps the most interesting cases involve component event handlers, method calls, and property getters & setters, because these node types allow abstracting over different component types with generic blocks. These use the *type* of the component (rather than its name) but include the name of the property or method. So the node type of `set Brick1.Enabled` is `set ImageSprite.Enabled` and the node type of `when Brick1.CollidedWith` is `event ImageSprite.CollidedWith`.

Other cases are straightforward, except that a procedure call node type includes the procedure name, and a local variable declaration node type includes the declared variable names.

With this notion of AST equivalence, the `CollidedWith` handlers like the one in Fig. 2 for all 30 bricks in the brick-breaker app are in the same equivalence class. Note that equivalence would still hold if some bricks were worth more points or updated a different label from `points`. This makes sense intuitively, because a procedure whose arguments include the brick component, the point value, and the point label component could still abstract over all the handlers.

B. Results on App Inventor Datasets

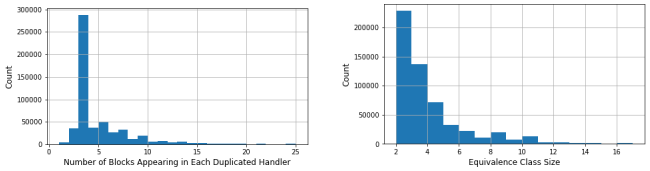
We applied the event handler equivalence algorithm to 10K random and 46K prolific datasets described in Sec. II. Some key results are summarized in Tab. I, in which equivalence classes are restricted to only *nontrivial classes* with more than one handler — i.e., each such class is an opportunity to abstract over code duplicated in multiple handlers.

A few results stand out. Event handler duplication was common in both datasets: 28.9% of 10K programmers and 95.6% of 46K programmers had a least one event handler duplication; and 21.2% projects in the 10K dataset and 20.2% of projects in the 46K dataset exhibited such duplication.

Feature	10K random	46K prolific
number of programmers	10,000	46,320
number of projects analyzed	30,851	1,545,284
number of screens analyzed	33,381	1,773,251
duplicated handler equivalence classes	12,932	558,944
median, avg, max handlers in class	3, 4.1, 88	3, 3.8, 80
median, avg, max blocks in duplicated handler	3, 8.2, 1,811	3, 5.0, 86
pct of equiv classes with proc call body	2.7%	5.5%
pct of equiv classes requiring generics	36.0%	33.2%
pct projects with duplicated handlers	21.2%	20.2%
pct programmers with duplicated handlers	28.9%	95.6%

TABLE I
RESULTS SUMMARY FROM EVENT DUPLICATION STUDY

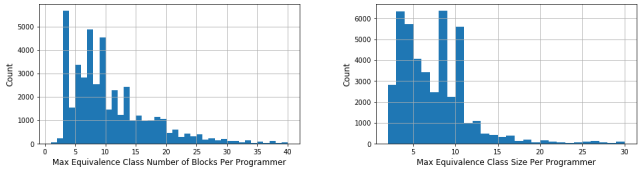
The number of blocks in each of the duplicated handlers tended to be very small (median of 3 in both groups), though there was a remarkable outlier of 1,811 blocks in the 10K group. Moreover, the size of each equivalence class (i.e., the number of duplicated handlers it contains) also tended to be small. Fig. 5 shows histograms of these two statistics for the 46K dataset (ignoring outliers), which includes only the nontrivial equivalence classes (i.e., number of handlers ≥ 2); distributions for the 10K dataset were remarkably similar.



a b

Fig. 5. Distributions in the 46K dataset for (a) the number of blocks in duplicated handlers and (b) the number of duplicated handlers in nontrivial equivalence classes.

A different picture emerges in Fig. 6, which shows the distributions of the *maximum* of these statistics for the subset of programmers having nontrivial equivalence classes (i.e., those counted in the last row of Tab. I). Such programmers tend to be impacted in at least one project by handler duplication where the blocks per duplicated handler and/or equivalence class sizes are larger than suggested by Fig. 5.



a b

Fig. 6. Distributions in the 46K dataset for programmers with at least one duplicated handler for (a) the max number of blocks in the handlers of an equivalence class and (b) the max number of handlers in the equivalence class.

About 2.7% of 10K and 5.5% of 46K nontrivial classes "do the right thing" in terms of having a handler that consists only of a call to a procedure, presumably one that abstracts over other duplicated code for the handler, as shown in Fig. 4.

Based on the equivalence relation, abstracting over the handlers in an equivalence class will require generic blocks

Percentages	10K random	46K prolific
nongeneric method, getter, & setter blocks	40.3%	44.6%
generic method, getter, & setter blocks	0.014%	0.21%
projects using generic blocks	5.0%	5.1%
programmers using generic blocks	9.2%	57.1%

TABLE II
STATISTICS INVOLVING THE USE OF GENERIC BLOCKS.

when a node has a component node type (such as `set ImageSprite.enabled`) that is instantiated to different components in different members of the equivalence class (such as `Brick1` and `Brick2`). A generic block is the only way to abstract over this difference. Note that 36.0% of handlers in the 10K dataset and 33.2% in the 46K dataset require generic blocks. This is an important result, because generic blocks are used relatively rarely in the wild (Tab. II). Whereas specific component method calls and property getters & setters account for (40.3%, 44.6%) of all blocks in the programs of (random, prolific) users, generic versions of these blocks account for only (0.014%, 0.21%) of all blocks, and such blocks are used only in about 5% of projects in both groups. This may be because programmers do not know about them (previously, they were very poorly documented and did not appear in standard tutorials) or find them difficult to use. So it is not surprising that code duplication is common in situations where generic blocks would be required to remove it. While only 9.2% of random users have some project using generic blocks, 57.1% of prolific users do, suggesting that most of them are eventually exposed to these blocks.

The results reported here differ substantially from those reported in an earlier version of this work [4]. The main reason is a difference in assumptions; the earlier work assumed it was not worthwhile to create procedures that had fewer than five blocks in their body. In this study, it is expected that generic event handlers will be used as the abstraction mechanism instead of procedures, and there *is* a benefit for using these even in situations where the handler body size is small.

V. GENERIC EVENT HANDLERS

A. Motivating Example

Since the launch of MIT App Inventor in December 2013, the MIT App Inventor team was aware of the inelegance of of abstraction pattern illustrated in Fig. 4, where a procedure captures the body pattern of similar event handlers, but the handler itself must be copied for each component. Also, several users have requested a more elegant mechanism to deal with this problem. But it wasn't until the precursor of the study described in Sec. IV that it became clear how pervasive the problem was and how important it was to solve it.

The study inspired the first author to recently extend App Inventor with **generic event handlers**, a new feature that makes it possible to eliminate the cumbersome and inelegant duplicate-handler-with-procedure-call in Fig. 4. For example, Fig. 7 shows a single generic event handler for `ImageSprite` components that handles all 30 bricks, *without* a procedure like `handleBallCollisionWith` in Fig. 4 or distinct handlers for each brick. When an event E occurs on a component

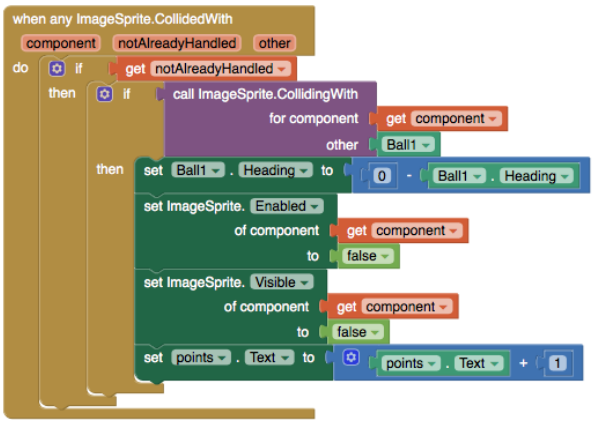


Fig. 7. A single `ImageSprite` handler that will handle all 30 bricks. The `component` parameter is automatically instantiated to all `ImageSprite` components, which include the 30 brick components and the paddle component. The `notAlreadyHandled` parameter is a boolean value that is `true` if and only if a more specific handler for the component does not exist. The project contains a specific handler for the paddle (not shown here), so this generic handler will not apply to the paddle.

C (e.g., `Brick1`), if there is a specific handler H_S for event E on C (e.g., `when Brick1.CollidedWith`), handler H_S's code runs. Then, if there is a generic handler H_G for event E on the type of component C (such as `when any ImageSprite.CollidedWith`), H_G's code runs. The `component` parameter holds a reference to the component, so it serves the same purpose as the `brick` parameter in the `handleBallCollisionWith` procedure in Fig. 4.

The `notAlreadyHandled` parameter is a boolean value that is `true` if and only if a specific handler for the event on `component` does not exist. In the case of the `when any ImageSprite.CollidedWith` handler, it is used in a conditional test to control *which* `ImageSprite`s are controlled by this handler. In particular, in the brick-breaker game, the paddle is also an `ImageSprite` component, but it should not be treated like a brick; although the ball should bounce off the paddle, that should not cause the paddle to disappear or the score to be incremented. The specific handler in Fig. 8 causes `notAlreadyHandled` to be `false` in the generic handler in Fig. 7, so the brick-specific code is not executed for the paddle.

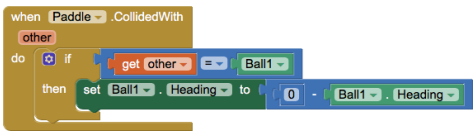


Fig. 8. The `CollidedWith` handler for the paddle in the brick-breaker game. This specific `CollidedWith` handler causes `notAlreadyHandled` to be `false` in the generic `CollidedWith` event handler for `ImageSprite`s in Fig. 7.

B. Associating Extra Information with Components

When the body of an event handler is proceduralized, as in Fig. 4, in general there could be many arguments to the procedure, which might or might not include the component. For example, Fig. 9 shows some core blocks of a xylophone app from an introductory App Inventor textbook [11]. Each of eight xylophone keys is represented as a rectangular colored button that, when pressed, both plays a note and records it for

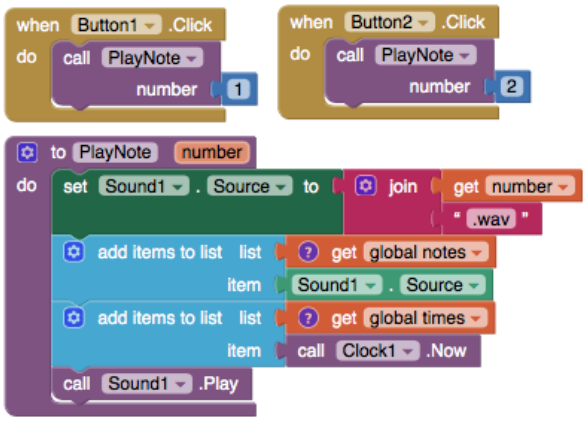


Fig. 9. Core blocks in a textbook xylophone app in which each key is represented as a button that both plays a note and records it for later playback. Only two of the eight `Button.Click` handlers are shown. Each just calls the `PlayNote` procedure with a number for the corresponding note to be played and recorded.

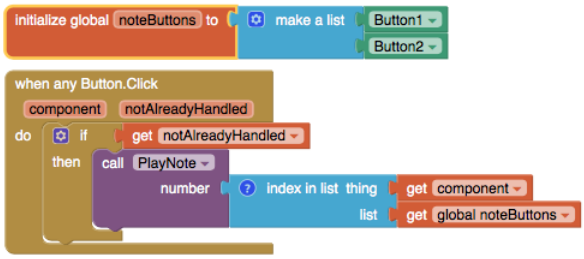


Fig. 10. One way to specify a single generic event handler to replace the xylophone key button handlers in Fig. 9. The note number is determined from the component by looking up its (1-based) index in a global list of button components. Only two of the eight buttons is shown in the list.

later playback. Notes are stored in audio files named `1.wav` through `8.wav`. The `PlayNote` procedure takes a note number as its only argument, sets the `Source` property of a `Sound` component to the corresponding audio file, records the note and timing information in two global lists, and finally plays the audio file. Each of the eight button handlers (for brevity only two are shown in the following examples) just calls the `PlayNote` procedure with an appropriate numeric argument.

There are a multitude of ways to replace the eight specific xylophone key button handlers with a single generic handler. All involve associating the correct note information with each key button component in such a way that it can be determined from the component. One approach is shown in Fig. 10, which determines the note number for a key button by looking up its (1-based) index in a global list `noteButtons` of all the key buttons. The `notAlreadyHandled` conditional is essential since the app also has playback and reset buttons whose behavior is controlled by specific button handlers. The `PlayNote` procedure is no longer really necessary; its code blocks could be inlined within the `when any Button.Click` generic handler, but we keep it for simplicity.

This approach leverages a clever choice of audio file names involving numbers. More generally, the note files might have names involving `LowC`, `D`, `E`, `F`, `G`, `A`, `B`, and `HighC`. In that case, the depicted solution could be adapted to have a parallel global list of note file names, and the key button index could

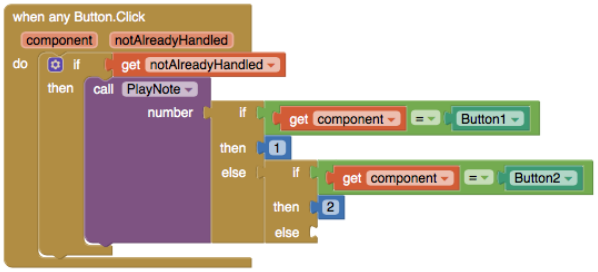


Fig. 11. This variant of Fig. 10 uses nested conditional expressions to map each button component to its corresponding note value. The empty `else` socket would be filled with more blocks to handle the rest of the buttons.

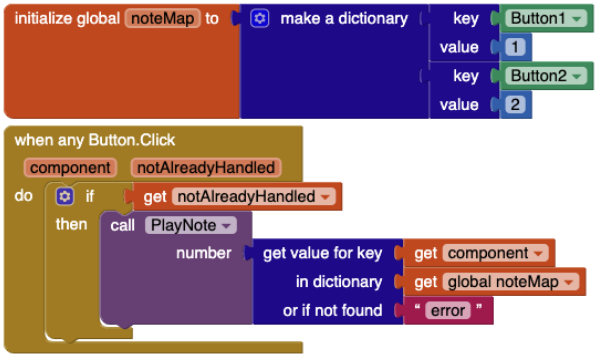


Fig. 12. This alternative to Fig. 10 uses a dictionary to map each key to its corresponding note value.

be used to extract the corresponding note file name.

There are many other ways to associate arbitrary extra information with components:

- Use nested conditional expressions that return the information for each specific component based on an equality test with the `component` parameter of the generic event handler. This is illustrated in Fig. 11.
- Create a global association list of component/info pairs, where each pair is itself a two-element list. App Inventor is equipped with a `look up in pairs` block that performs the desired lookup.
- App Inventor does not yet support a dictionary datatype, but there are plans to add one [12]. Dictionaries could model any association between a component and additional information, as shown in Fig. 12. While the version reported in [12] was constrained to only string keys, that restriction has been relaxed to allow components as keys.
- App Inventor can be extended so that each component has a special `Datum` property that allows any piece of information to be associated with that component (Fig. 13). Since the datum can be a list or (eventually) a dictionary, in practice any number of pieces of information can be associated with a component.
- A variant of associating a single `Datum` property with each component is for every component to allow any number of user specified settable and gettable properties, in the spirit of MacLisp's `putprop` and `get` operations on symbols, which supported a early kind of object-based programming. Fig. 14 shows how the semantics of custom component properties might be realized in App Inventor.

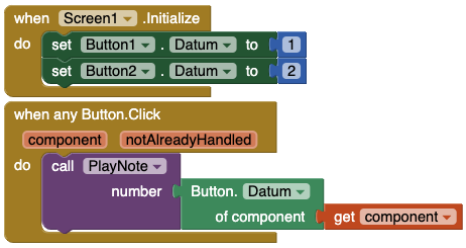


Fig. 13. This variant of Fig. 10 uses a datum associated with each button to map it to its corresponding note value.

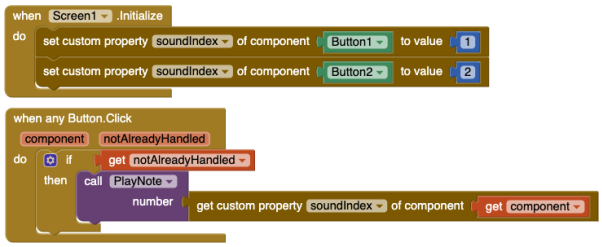


Fig. 14. This Fig. 10 variant uses custom component properties similar to MacLisp's `get`/`putprop` functionality to associate note values with buttons.

C. Leveraging Component Names

Naming variables based on their purpose or function is widely considered to be best practice in software engineering as it aids in readability and maintainability of code. Because component names could hold useful information about the functionality of the system, one option would be to extend the language so that programmers could obtain the name of a component as a string or obtain a component given its name. Many of the examples in Section V-B include code to establish a mapping between a component and relevant data. This manual construction can be abstracted away in the form of code that performs a substitution so that mapping need not be updated as the app is extended. This approach, called *coding by convention*, is used in some domain specific languages, such as Grails (http://docs.grails.org/latest/guide/single.html#conventionOverConfiguration). It serves as another abstraction that users can employ to reduce code complexity through the principle of *Don't repeat yourself* (DRY).

For example, consider a version of the xylophone app in which the buttons are named `Low_C_key`, `D_key`, etc. By adding the ability to access the names of the components in the language, the suffix `_key` can be replaced with `.wav` to construct the file name of the corresponding sound file. Adding a new sound simply requires appropriately naming a new button. Alternatively, if a `Player` component is added corresponding to each `Button`, one could dynamically map `Button`s to `Player`s by making name substitutions at runtime. Fig. 15 shows an example of repurposing the integer in a xylophone button name to play a note.

VI. Specific/Generic Conversion

Manually converting the specific event handler in Fig. 2 to the generic event handler in Fig. 7 is a tedious process requiring many steps. Specific component method calls and property getters/setters must be disconnected from the event handler, corresponding generic blocks must be selected from a blocks menu drawer, argument block subassemblies need to

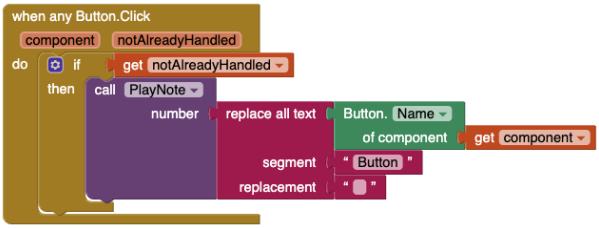


Fig. 15. A variant of Fig. 10 where data for the app logic (in this case the note number) is extracted from the button name rather than using an explicit mapping. App Inventor treats strings containing only digits as numbers. The green `Button.name of component` block does not yet exist in App Inventor, but is being considered.

be moved from the specific blocks to the generic blocks, the generic blocks must be connected back to the event handler (possibly with conditionals to distinguish subcategories of components), and the specific blocks need to be deleted.

In a cognitive perspective on notation, cumbersome notational changes like this are said to have *high viscosity* [13], [14]. Reducing viscosity is sometimes explicitly mentioned as a design goal for editing features in blocks languages (e.g. [15]) and closely-related structure editors, like Stride's frame-based editor [16].

In addition to the primary goal of supporting better event handler abstraction, a secondary goal of our work is to reduce the viscosity of **genericization**, which is the process of converting code that uses specific component blocks into code that uses generic blocks. Towards this end, we modified the App Inventor environment to include block context menu items that allow users to easily convert between specific and generic component methods and property setters & getters. For example, clicking the `Make Generic` context menu option for the nongeneric blocks `Brick1.CollidedWith`, `set Brick1.Enabled`, and `set Brick1.Visible` in Fig. 2 would yield the corresponding generic blocks in Fig. 3. Similarly, clicking the `Make Specific` context menu option for the three generic blocks in Fig. 3 would convert them back to the corresponding nongeneric blocks in Fig. 2.

There is even a `Make Generic` context menu option for the `Brick1.CollidedWith` handler block in Fig. 2 that will automatically convert it to the blocks in Fig. 16, which is almost the same as the blocks in Fig. 7 except for the missing `if notAlreadyHandled` wrapper. Because there can be arbitrary conditionals within a generic event handler that depend on the desired semantics for the given project, such conditionals must be manually inserted by the programmer and cannot generally be automatically deduced. For a similar reason, there is no `Make Specific` context menu option for generic event handlers.

The `Make Generic` context menu option facilitates genericization. We hypothesize that this menu option will help users generalize handlers, procedures, and block assemblies involving particular components to more abstract versions that can involve some subset of (possibly all) components of a particular type. We also hypothesize that the `Make Generic` and `Make Specific` context menu options will raise awareness about generic vs. nongeneric blocks, which, as noted earlier, are rarely used in the wild. However, user studies

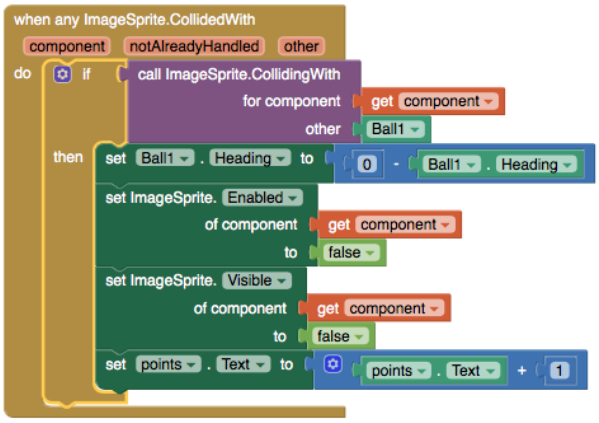


Fig. 16. This handler is the same as the one in Fig. 7 *except* that it does not include the `if notAlreadyHandled` wrapper around the remaining body blocks of the handler. This is the result of selecting the `Make Generic` option for the handler in Fig. 2.

and analyses of projects edited with the enhanced system will need to be undertaken to evaluate these hypotheses.

VII. Discussion and Future Work

Thunkable (https://thunkable.com/) is a commercial version of App Inventor with some advanced features not in MIT App Inventor. The Thunkable team independently developed a version of generic handlers. They do not use a `notAlreadyHandled` flag to control the application of generic vs. specific handlers for a component. Instead, all handlers for a component are run in parallel by interleaving execution of the code in their bodies; within a generic handler, conditional tests on component values can be used to control the behavior of specific components. Thunkable also supports custom user properties for components, which allows associating data with components along the lines shown in Fig. 14. Thunkable does not have a feature for converting between specific and generic blocks.

Our work has focused on a particular kind of code duplication in App Inventor: event handlers whose code bodies have ASTs with the same shape. Anecdotally, we have observed other kinds of code duplication we plan to investigate in the future, including: (1) event handler bodies whose ASTs do not have the same shape but still have large chunks of similar code that is repeated and could be captured by procedures; (2) long sequences of sequential code with repeated patterns than can be captured by a combination of lists, loops, and procedures; and (3) multiple screens that can be abstracted by a single screen that instead reads information from files.

Other future work for this project includes educating the community about our new generic features, studying their use in the wild, and exploring interventions that encourage using them to avoid code duplication. In particular, inspired by the work of Techapalokul and Tilevich on refactoring in Scratch [17], [18], we plan to experiment with different approaches for automatically refactoring App Inventor programs to eliminate code duplication through a combination of generic handlers, procedures, loops, lists, and files. For example, if a user's project has code duplication, the duplicates could be highlighted, and the user could be given various options, ranging from "Automatically remove this duplication for me" to "Teach me the steps for removing this duplication so that I know how to do it on my own in the future". We also plan to explore the detection of repeated copy/pasting of the same code and present the option to refactor instead, for example by using a loop over a list and/or introducing new procedures. Our refactoring plans for introducing generic handlers and procedures correspond to *Extract custom block*, one of four refactoring techniques for Scratch in [18]. Because App Inventor is not constrained by certain limitations of Scratch (e.g., custom blocks cannot return values), we expect that this technique will have wider applicability in App Inventor.

More broadly, users have goals beyond the pedagogical goals of MIT App Inventor. Building complex projects without good tooling can result in discouragement and project abandonment. One example in our data set was a calendar app with a handful of button click handlers implemented, some in a partial state of completion, and most without any functionality. The user went through much trouble laying out the app interface only to hit a wall when implementing the logic of the many buttons required to create the calendar. Better tools might have decreased friction, increased the user's engagement, and encouraged them to continue the project.

VIII. Conclusion

Our analysis of 1.5 million App Inventor projects by prolific users (those with 20 or more projects) shows that nearly 96% of them have at least one project containing code duplication in event handlers. Based on this observation, we developed generic event handlers to allow users to handle events occurring on any component of a given type. Further, we added functionality to automatically convert blocks code from specific references into the equivalent code using generic blocks (with references to the specific component) to aid in manual refactoring of these code blocks.

We proposed five alternative methods for associating data that might be needed for more sophisticated refactoring–*conditionals*, *associative lists*, *dictionaries*, *component datum*, *user props*–and discussed the potential benefits and drawbacks of each. Future work will include evaluating these different approaches in a classroom setting with high school students.

Lastly, this **data-informed programming language design** approach suggests that we can discover opportunities to improve visual programming languages, such as App Inventor, through data-driven analysis of projects to reduce the friction encountered by users trying to accomplish their goals.

Acknowledgments

Seo's research was funded by the 2018 Wellesley College Science Center Summer Research program through the IBM Faculty Research Fund for Science and Math and by a Wellesley College Faculty Grant. The App Inventor dataset was provided by the MIT App Inventor team's Jeff Schiller. Some of our analyses use a Python project summarization

program that builds upon earlier work by Isabelle Li, Maja Svanberg, and Benji Xie.

REFERENCES

[1] 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)*, pp. 53–61, Sept. 2016.

[2] P. Techapalokul and E. Tilevich, "Understanding recurring quality problems and their impact on code sharing in block-based software," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '17)*, pp. 43–51, Oct. 2017.

[3] I. Li, F. Turbak, and E. Mustafaraj, "Calls of the wild: Exploring procedural abstraction in App Inventor," in *2017 IEEE Blocks and Beyond Workshop*, pp. 79–86, Oct. 2017.

[4] A. Seo, "Abstractionless programming in App Inventor," in *BLOCKS+ 2018 ACM SPLASH Workshop*, Nov. 2018.

[5] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[6] H. Keuning, B. Heeren, and J. Jeuring, "Code quality issues in student programs," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*, pp. 110–115, July 2017.

[7] D. Wolber, H. Abelson, and M. Friedman, "Democratizing computing with App Inventor," *GetMobile: Mobile Computing and Communications*, vol. 18, pp. 53–58, Oct. 2014.

[8] B. Xie and H. Abelson, "Skill progression in MIT App Inventor," in *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '16)*, pp. 213–217, Sept. 2016.

[9] 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*, pp. 1–8, ACM, Oct. 2015.

[10] F. Turbak, M. Sherman, F. Martin, D. Wolber, and S. C. Pokress, "Events-first programming in App Inventor," *Journal of Computing Sciences in Colleges*, vol. 29, pp. 81–89, Apr. 2014.

[11] D. Wolber, H. Abelson, E. Spertus, and L. Looney, *App Inventor 2: Create your own Android Apps*. O'Reilly Media, Inc., Oct. 2013.

[12] E. W. Patton and D. Tang, "JSON interoperability in MIT app inventor," in *BLOCKS+ 2018 ACM SPLASH Workshop*, Nov. 2018.

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

[14] 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.

[15] F. Turbak, D. Wolber, and P. Medlock-Walton, "The design of naming features in App Inventor 2," in *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '14)*, pp. 137–144, Aug. 2014.

[16] M. Kölling, N. C. C. Brown, and A. Altadmri, "Frame-based editing," *Journal of Visual Languages and Sentient Systems*, vol. 3, pp. 40–67, July 2017.

[17] P. Techapalokul and E. Tilevich, "Programming environments for blocks need first-class software refactoring support," in *2015 IEEE Blocks and Beyond Workshop*, pp. 109–111, Oct. 2015.

[18] P. Techapalokul and E. Tilevich, "Code quality improvement for all: Automated refactoring for scratch," in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '19)*, pp. 117–126, Oct. 2019.