Live programming of mobile apps in App Inventor

Jeffrey Schiller
Hal Abelson
José Dominguez
Andrew McKinney
MIT
{jis,hal,josed,mckinney}@mit.edu

Franklyn Turbak
Johanna Okerlund
Wellesley College
{fturbak,jokerlun}@wellesley.edu

Mark Friedman
Google
markf@google.com

Abstract

MIT App Inventor is a programming environment that lowers the barriers to creating mobile apps for Android devices, especially for people with little or no programming experience. App Inventor apps for a mobile device are constructed by arranging components with a WYSIWYG editor in a computer web browser, where the development computer is connected to the device by WiFi or USB. The behavior of the components is specified using a blocks-based graphical programming language. A key feature in making App Inventor accessible to beginning programmers is *live programming*: Developers interact directly with the state of the evolving program as it is being constructed, and changes made in the web browser are realized instantaneously in the running app on the device. This paper describes the live programming features of App Inventor and explains how they are implemented.

Categories and Subject Descriptors D.1.7 [*Programming Techniques*]: Visual Programming

General Terms mobile app development, visual languages, liveness, live programming, live coding, debugging, software development tools

Keywords mobile app development, live programming, Android, interpretation

1. Introduction

In a *live programming* environment, code changes are immediately and continually reflected in a constantly running program. Liveness makes program development more interactive by incorporating the effect of program edits more quickly than the traditional edit-compile-run-test approach.

Here we describe the live programming capability of MIT App Inventor [17], an environment for developing mobile apps for Android smartphones and tablets.[1] The App Inventor environment runs in a web browser where the user specifies the components and behavior of an Android app to be executed on a real or emulated Android device. The App Inventor source code is available under an open source license [18], so that anyone can deploy App Inventor servers. MIT also operates a large public service that hosts 87,000 users weekly, with a total of 2.2 million registered users, who among them have developed 5.5 million apps [17]. App Inventor's main goal is to democratize mobile app development by giving people with little or no programming experience appropriate app-building tools, empowering them to become app producers rather just downloaders.

Some App Inventor features designed to lower barriers to app creation for novices are:

- Apps are assembled from *components* that encapsulate features of the Android SDK. Each component advertises a collection of *methods*, *properties*, and *events*. The components of an app, including the arrangement of its graphical user interface elements, are specified using a WYSIWYG drag-and-drop editor.
- Components' behaviors are specified in a blocks programming language, in which fragments shaped like jigsaw-puzzle pieces are connected to form programs. Blocks shapes eliminate frustrating syntax errors. Menu-based naming features reduce errors like misspelled names and unbound variables [28], and menu-based drawers of related blocks with labeled sockets help programmers choose the correct blocks and remember the number and order of their operands.
- Blocks provide access to high-level behaviors, making it easy to build apps that incorporate Android device features like touch-based interfaces, phone calls and texting, location awareness, accelerometer sensor, orientation sensor, camera, sound recorder, speech synthesis and speech recognition, language translation, social media, persistent data storage, cloud-based data storage, sprite-based gaming, and interacting with web services. Many simple but compelling apps can be created with fewer than ten blocks, and nontrivial useful apps can be created with just a few dozen blocks [29].

Another key feature of the App Inventor environment is that apps are typically created using *live development mode*, where the environment in the web browser is connected to a running app on an Android device, and changes made to the app's user interface or code blocks are immediately reflected on the device. There is also an ability (*DoIt*) to immediately execute any block in the context of the running app and see the value, if any, produced by that execution. Such live development enhances the experience of creating, testing and debugging apps by eliminating the tedious edit-build-install-test cycle traditionally associated with mobile app develop-

[1] The implementation described here is *App Inventor 2*, which was released in December 2013. The predecessor *App Inventor Classic* system was released in March 2012. App Inventor Classic is in turn based on an earlier system developed at Google Research from 2008–2012.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

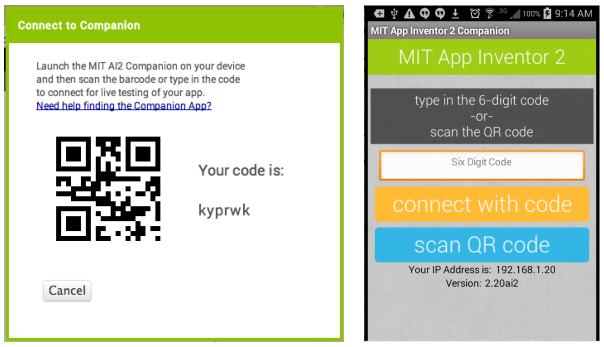
ment. New and seasoned App Inventor programmers alike often remark that live development mode is one of the most important features of App Inventor. Using the browser to interactively develop and test a running app on an Android device connected via WiFi is an experience that many users describe as "magical."

In this paper, we explain how the "magic" of live development is achieved. The key is a special app running on the target device, the *App Inventor Companion*. Although apps can ultimately be compiled to produce ordinary `apk` files, browser interaction during live development is accomplished by the Companion, which serves as an interpreter for the App Inventor language. The Companion also manages the WiFi connection between browser programs and external devices with the aid of a *rendezvous server* and enables powerful debugging tools that allow users to probe the running app on the mobile device.

2. App Inventor Live Development Example

We introduce the live programming features of App Inventor in the context of building a simple app in which a ball is flung with a finger and bounces off the edges of an enclosing canvas.

In App Inventor, we build an app in a web browser on a computer that we connect to an Android device. We begin by visiting `http://ai2.appinventor.mit.edu` in the browser and starting a new project. This puts us in the Designer window for the app, where we specify its user interface components (in this case, a canvas and ball) and behavioral components (there are none initially, but later we will modify the app to have a clock component).



(a) (b)

Figure 1. Steps in connection App Inventor to an Android device.

To illustrate live development, we will connect the browser to the device via WiFi before adding any components to the app. First, in the browser, we select *Connect>AI Companion*, which displays a six-character code and its corresponding QR code (Fig. 1a). Next, on the device, we launch the MIT AI2 Companion app, which we have downloaded from the Google Play Store. This presents an interface in which we can either type in the six-character code or, equivalently, scan the QR code (Fig. 1b). Once we do so, the AI2 development environment in the browser and the device are "connected", and changes to the app made in the browser are reflected on the device. Initially, the device shows an empty screen labeled *Screen1* because the app has no components yet (Fig. 2a).

In the Designer, we add a `Canvas1` component to the app by dragging it from the *Drawing and Animation* palette onto a representation of *Screen1* within the Designer. In the Properties pane, we edit the properties of `Canvas1`: set its color to cyan (Fig. 2b), width to 200 pixels (Fig. 2c), and height to 300 pixels (Fig. 2d). Each property edit is reflected both in the representation of *Screen1* in the Designer and on the actual device screen.

We add a ball sprite `Ball1` to the canvas by dragging it from the *Drawing and Animation* palette. The ball immediately appears

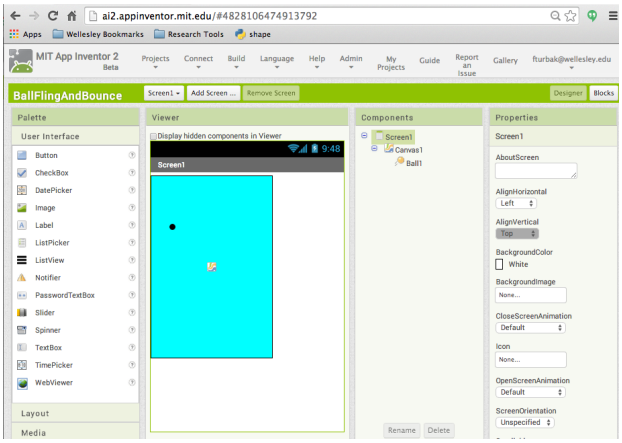


Figure 3. Designer after `Canvas1` and `Ball1` have been added.

within the canvas on the device (Fig. 2e). At this point, we are done creating the app's user interface in the Designer (Fig. 3).

To make the ball move, in response to flinging it with a finger, we go to the Blocks Editor in the browser, and assemble a `Ball1.Flung` event handler from blocks selected from various drawers of blocks (Fig. 4). Once this is done, the ball on the canvas moves when flung. We do not have to activate the event handler by compiling or sending it to the device; the mere presence of an event handler in the Blocks Editor makes it active on the device.

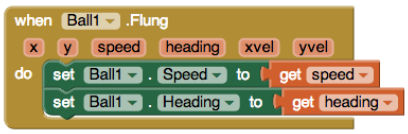


Figure 4. Blocks event handler for flinging `Ball1`.

We can make the ball larger by assembling the blocks in Fig. 5 and selecting the *DoIt* option in the context-sensitive menu for the `set Ball1.Radius` block. This immediately changes the ball's radius on the device, even if the ball is currently moving.

Figure 5. Selecting the *DoIt* option on these blocks immediately changes `Ball1`'s radius, even if it is moving.

Suppose we want to show the path taken by the ball. In the Designer, we can add a Clock component from the Sensor palette that has a timer interval of 400ms, and then in the Blocks Editor assemble the blocks in Fig. 6. This causes a red dot to be drawn on the canvas at the center of the ball every 400ms, thus showing the path taken by a moving ball. If the `Clock1.Timer` blocks are assembled while the ball is moving, the dots will be drawn as soon as the event handler is fleshed out. An example of the path taken by a flung ball is shown in Fig. 2f.

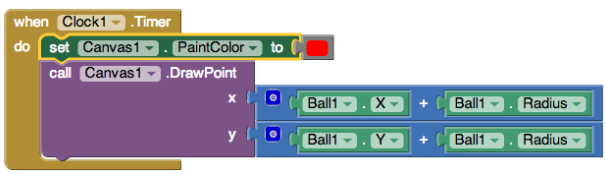


Figure 6. An event handler that draws a red dot at `Ball1`'s center every time the timer fires.

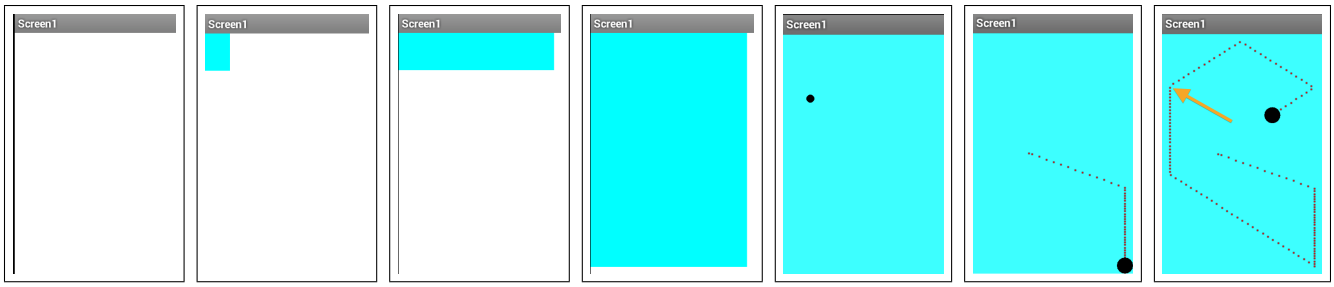


Figure 2. Android device screen shots from intermediate steps of ball-in-canvas example.

When ball hits the right edge of the canvas, it hugs the edge until it gets stuck in the bottom corner. This is because it can't move past the right edge but still has a downward velocity component. We can change this behavior by adding the event handler in Fig. 7, which makes the ball bounce off an edge that it hits. When this event handler is completed, the bouncing behavior takes effect immediately. For example, suppose we fling the ball out of the bottom right corner, and we finish the handler as the ball is hugging the left canvas edge. As shown in Fig. 2g, the ball will first bounce off the left edge at the point (to which the green arrow points) when the `Ball1.EdgeReached` handler is completed. The ball will continue bouncing off other edges afterwards.

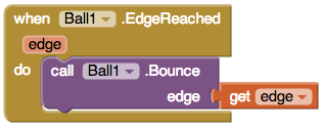


Figure 7. An event handler that causes `Ball1` to bounce off the edges of the canvas.

We have shown live app development with complete event handlers, but the device executes even partially defined handlers. E.g., in the `Clock1.Timer` handler, if the blocks for the x argument of `Canvas1.DrawPoint` are missing, the handler will still execute, but the Blocks Editor on the browser will report a missing argument error. This error will not stop the motion of the ball; it will just stop the drawing of the dots. This underscores the liveness of the system — any code on the screen within an event handler is executed on the device when the corresponding event occurs, even if the handler is incomplete.

3. Implementing Liveness in App Inventor

3.1 App Inventor System Architecture

App Inventor has four main architectural components (Fig. 8). Two of these — the web browser with the App Inventor environment and the Android device on which live development is performed — were introduced in the above example. The device can be a physical device, such as a phone or a tablet, or it can be a virtual device such as the Google-distributed Android emulator.

A third component is the App Inventor server, which provides the App Inventor environment implementation downloaded by the browser and also hosts cloud-based storage for user projects.

3.2 Packaging Applications with YAIL, Kawa, and Forms

App Inventor also has a fourth component that does not participate in live programming. This is the build server, which compiles a complete App Inventor program into an Android packaged application (`apk` file) suitable for distribution, including distribution

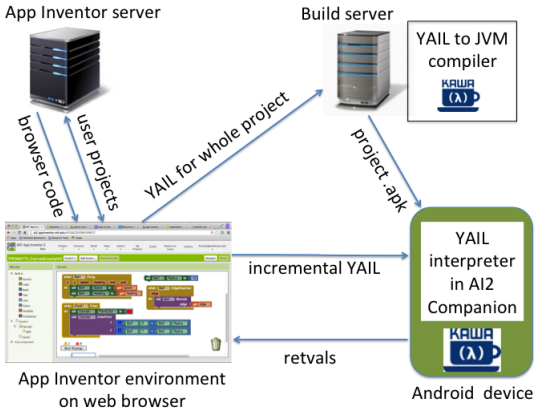


Figure 8. The key elements and interactions in the App Inventor system. The browser and Android device are assumed to be connected and communicating as explained in Sec. 4.

through the Google Play App Store. When a user invokes the *Build* option, App Inventor uploads the complete project to the build server; the resulting app can be downloaded to the device via a QR code scanned from the browser or downloaded as a file to the user's local computer for further distribution.

In order to create a packaged application, the components and blocks for each screen of an App Inventor project are first converted to YAIL (Young Android Intermediate Language), a language defined as a collection of macros and functions in Kawa [3], a Java-based implementation of Scheme [24]. Kawa was chosen by the original App Inventor implementers [8] because it combines Scheme's flexibility for defining embedded languages with the ability to integrate with Java libraries and compile to Java Virtual Machine (JVM) bytecodes, and it had been demonstrated as a framework for building Android apps [4]. The build server compiles YAIL code to JVM bytecodes, which are then converted to Dalvik Virtual Machine bytecodes when creating the `apk` file.

For instance, Fig. 9 shows YAIL code that creates the canvas and ball and defines the flinging handler for the example from Sec. 2. `define-form` creates an instance of `Form`, a Java class in the App Inventor implementation that is a subclass of Android's `Activity` class, a fundamental unit of an Android app. `add-component` creates an App Inventor component (also an instance of a Java class, in this case one that abstracts over an Android device feature) and adds it to a form; its properties are changed by `set-and-coerce-property!` `define-event` creates an event handler registered with the current form. `init-runtime` connects the Android event handler to the App Inventor event handlers.

App Inventor users are often surprised to learn that Java source code is never generated during the implementation process. In particular, the blocks code is never converted to Java.

```
(define-form appinventor.ai_testuser.BouncingBall.Screen1
     Screen1)

;;; Screen1
(do-after-form-creation
  (set-and-coerce-property! 'Screen1 'Title "Screen1" 'text))

;;; Canvas1
(add-component Screen1 Canvas Canvas1
  (set-and-coerce-property! 'Canvas1 'BackgroundColor
       #xFF00FFFF 'number)
  (set-and-coerce-property! 'Canvas1 'Width 200 'number)
  (set-and-coerce-property! 'Canvas1 'Height 300 'number))

;;; Ball1
(add-component Canvas1 Ball Ball1
  (set-and-coerce-property! 'Ball1 'X 46 'number)
  (set-and-coerce-property! 'Ball1 'Y 27 'number))

(define-event Ball1 Flung($x $y $speed $heading $xvel $yvel)
  (set-this-form)
  (set-and-coerce-property! 'Ball1 'Speed
    (lexical-value $speed) 'number)
  (set-and-coerce-property! 'Ball1 'Heading
    (lexical-value $heading) 'number))

(init-runtime)
```

Figure 9. Some YAIL code for the example in Sec. 2.

3.3 Interactive YAIL Interpretation in the Companion

Beyond compilation and packaging apps, YAIL-in-Kawa is also the essence of live programming in App Inventor. The reason is that Kawa provides an interpreter that functions in a Read/Eval/Print loop (REPL). A REPL is the main interactive construct of any interpreted language that provides an interactive shell for evaluating individual expressions (e.g., Scheme, Lisp, Perl, Python, PHP, and many Javascript implementations). In contrast, in languages without an interactive interpreter (such as C and Java), whole program units must be compiled before they can be executed.

The Companion App that an App Inventor user runs on the device during development contains a Kawa REPL that can evaluate individual YAIL expressions sent from the browser.[2]

This REPL is embedded within an instance of the `ReplForm` subclass of the `Form` class. An instance of `ReplForm` is itself an Android activity with access to the screen and all phone events; it also has access to the App Inventor implementation libraries (including all App Inventor components) and the Android libraries. So YAIL code executed within the Companion's REPL has access to all the runtime state of the phone and can potentially do anything that any App Inventor app can do. Given the right YAIL expressions as inputs, the Companion App can faithfully impersonate *any* single-screen App Inventor app.

Reconsider the example from Sec. 2 in this context. When the Companion App starts, it has an empty screen named *Screen1*. When the user adds a canvas to the app, sets the canvas properties, and then adds the ball, the browser sends a sequence of YAIL expressions to the Companion that in real time builds the user interface on the device as shown in Figs. 2b–2e, providing liveness in the construction of the app's GUI. Then, when the `Ball1.Flung` handler is completed, the the browser sends the following YAIL code to the Companion:

```
(process-repl-input 36 ; ID number of the handler block
  (define-event Ball1 Flung($x $y $speed $heading $xvel
       $yvel)
    (set-this-form)
    (set-and-coerce-property! 'Ball1 'Speed
      (lexical-value $speed) 'number)
    (set-and-coerce-property! 'Ball1 'Heading
      (lexical-value $heading) 'number)))
```

This defines and installs the flinging handler for `Ball1`, at which point the ball immediately becomes flingable.

Every time blocks change in the Blocks Editor, the YAIL code associated with all top-level declaration blocks (event handlers, procedure definitions, and global variable definitions) is regenerated. If the newly generated YAIL differs from a cached copy of the previously generated code, the browser sends the new YAIL to the device. This accounts for much of the liveness experienced by App Inventor programmers. For example, if we change `Ball1.Y` to `Ball.X` in Fig. 6, the browser will send a new `(define-event Clock1 Timer ...)` expression to the device with the new event body, and that body will execute the next time the timer fires. If we disconnect the blocks in `y` operand from `Canvas1.DrawPoint`, the brower will again send a new handler declaration, this time with an incomplete body. When `Canvas1.DrawPoint` is called with a missing argument, an exception is raised that is caught by a top-level exception handler in the REPL, and information about the error is packaged into a return value (*retval* for short) that is transmitted back to the browser (see Fig. 8). An appropriate error message is then displayed in the browser window.

Remarkably, the Companion App is itself an app created with App Inventor, but it does use a few special-purpose blocks that are normally hidden from regular users. Because it can behave like any App Inventor app, the Companion must request device permissions for all possible App Inventor apps. Also, since the Companion encapsulates the code for all App Inventor components and the App Inventor runtime system, any time one of these is modified (e.g., to add a new component feature or fix a bug) in a new release of the App Inventor environment, a new version of the Companion App is created, and this must be downloaded by all App Inventor users. This is handled through automatic updates of Google Play Store apps. MIT's distribution system for App Inventor also arranges for automatic updating of Companion apps that were not obtained through the Play Store.

3.4 Live debugging with *DoIt* and *Watch*

The REPL-based nature of the Companion App makes it easy to execute arbitrary blocks (not just top-level declarations) relative to the current state of the running app. This is the essence of a powerful debugging feature called *DoIt*. When the programmer selects a block and specifies *DoIt*, the browser simply generates the YAIL code for the block and sends that code to the Companion to be evaluated. For example, here is the YAIL generated by *DoIt* for the example in Fig. 5 (where 186 is the ID of the `set Ball1.radius` block):

```
(process-repl-input 186 (set-and-coerce-property! 'Ball1
     'Radius 10 'number))
```

When *DoIt* is performed on an expression block (a block that produces a result value, as indicated by a plug on its left-hand side), the REPL sends back to the browser a retval that associates a string representation of the resulting value with the ID of the block. The browser displays the value on the associated block. For example, Fig. 10, shows the result of invoking *DoIt* on the `+` block. No visual feedback is shown on non-expression blocks. Invoking *DoIt* on the `set global num` block in Fig. 10 changes the value of the global `num` variable to 42, but no value is displayed.

DoIt is a powerful debugging tool because it lets the programmer interactively probe the running app on the mobile device: an example of *live debugging*. It contributes to the liveness of App Inventor by helping users obtain a more immediate understanding of

[2] The note [8] describes how live programming came about somewhat serendipitously in App Inventor as a consequence of choosing Kawa as the implementation language. Live programming, for all its importance, was not part of App Inventor's original design.

the changing state of the running program. More generally, it adds to the sense of play and experimentation that is, or at least should be, particularly important to programmers, especially novices.

Figure 10. Invoking *DoIt* on an expression block displays the value of that block, as calculated by the interpreter on the device.

An extension to *DoIt* is *Watch*, which shows a sequence of *all* the values of a block, not just the most recent one (e.g., Fig. 11) [20]. This is implemented simply by tagging the YAIL expression for the block as a `watch` expression with its block ID. When the interpreter on the device encounters a `watch` expression, it evaluates the body, and sends back to the browser a retval tagged as a `watch` result with the block ID and value.

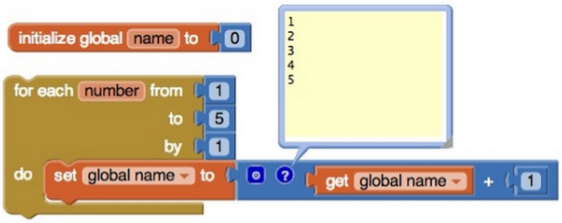


Figure 11. Selecting *Watch* on an expression within a loop shows all values of that expression when the loop is executed.

3.5 Apps with Multiple Screens

An App Inventor app may have multiple screens, only one of which is active (visible on the device) at any time. New screens are declared in the Designer. The `open another screen` block suspends the currently active screen, pushing it on a stack of suspended screens, and creates a new instance of its operand screen that becomes the new active screen. The `close screen` block deletes the current active screen, pops the top screen off the stack of suspended screens, and makes that the active screen. To allow interscreen communication, there are also versions of the screen opening and closing blocks that pass values between the screens.

Multiple screens interact with live development in two ways. First, the screen being edited in the browser is always synchronized with the active screen on the Companion. If the user changes screens in the browser, the YAIL for the new screen is sent to the Companion. And if the active screen changes in the Companion (as a result of executing `open another screen` or `close screen`), the Companion sends a special retval to the browser indicating that the Blocks Editor should display the blocks for the new active screen. This synchronization the Blocks Editor with the active screen of the Companion is essential for liveness. A lack of such synchronization would cause confusion — what would it mean to change blocks for *ScreenA* in the Blocks Editor if *ScreenB* were active on the Companion?

Second, the Companion's handling of multiple screens in live development mode is not exactly faithful to the behavior of multiple screens for packaged apps. In a packaged app, a suspended screen is a running `Form` instance (an Android activity) that can still process certain events unrelated to the user interface, like timer events. Also, when the top suspended app becomes active, its state is the same as when it was suspended (modulo changes made by events handled while it was suspended). In contrast, in live development, there is only one instance of `ReplForm` that attempts to simulate multiple screens, but this simulation is imperfect. In particular, (1) a suspended app never processes any events, and (2) when the top suspended app is activated, its suspension state is lost because all of its components and global variables are reinitialized. We can potentially increase the faithfulness of the Companion's simulation of multiple screens by saving a screen's state when it is suspended and reinstalling that state when it is activated. This would address problem #2, but addressing problem #1 (having suspended apps process events in live development) would be very challenging.

3.6 Liveness Issues with the Designer

In the current version of App Inventor, every change in the Designer (such as adding a component to the screen, deleting a component, renaming a component, or changing the initial value of a property) stops the current app being simulated by the Companion and reinitializes its user interface and program. From the viewpoint of liveness, this is undesirable.

For example, suppose the bouncing ball app from Sec. 2 is in the state shown in Fig. 2e, and we have added the `Ball1.Flung` event but have not added the `Clock1` component. Suppose we fling the ball. While the ball is in motion, we would like to be able to add the `Clock1` component and a `Clock1.Timer` event for tracing the ball's trajectory without stopping the program. However, adding the clock component stops the program and reinitializes its interface, including resetting the ball to its original position and setting its speed and heading to 0.

Reinitialization simplifies the implementation, but there is no deep reason why App Inventor could not be more clever and just modify the changed component without modifying the others. We plan to investigate this opportunity to improve liveness.

One issue is what users expect to happen in the running app when they change a property value within the Designer. Suppose `Ball1` is moving with a heading of 45 degrees the user changes the `heading` property of `Ball1` to 90 within the Designer. Does this mean that the user wants the moving ball's heading to immediately change to 90, or that they want its heading to be 90 the next time the app launches? Since they can already use *DoIt* to change the ball's current heading, it may be preferable for property changes in the Designer *not* to be reflected in the running app. This is a situation where an experimental study of user expectations may be in order.

4. Browser/Device Communication

The architecture for live development in Fig. 8 assumes two-way communication between the App Inventor environment running in a web browser and the Companion App running on an Android device. But how can we connect a web browser to an Android device and get them to communicate? This is a challenging technical problem whose solution we describe in this section.

4.1 Connecting the device with the browser

App Inventor runs in a web browser, so the logical approach for communication is to have the device act as a web server. The browser can then send YAIL to the device as an Asynchronous Javascript (AJAX) call, the same technology used by many web-based applications. Before AJAX calls can be made, the App Inventor environment needs to know the Internet Protocol (IP) address of the device to talk to. This is done in one of two ways. The App Inventor programmer can declare that they wish to connect to their device via a USB cable, or via a wireless (WiFi) connection.

If the device is connected via USB, then the IP address is a known quantity and communication can begin immediately. If a

WiFi connection is needed, a *rendezvous server* is used. When the programmer requests a wireless connection, the browser displays a randomly generated six-character code and its associated QR code (Fig. 1). When this code is entered on the device, the Companion (1) sends the code and its IP address to the rendezvous server and (2) launches a small web server listening on port 8001.

At the same time, the browser queries the rendezvous server, once per second, asking for the IP address for the displayed code. Until the device provides the address, the rendezvous server returns an empty response. However once it has learned the device's IP address, the rendezvous server can provide the device's IP address to the browser. Now the browser can use AJAX requests to deliver YAIL expressions to the Companion.

There are a few details that are needed to make this connection technique work reliably and securely. First, all communications in the protocol use the MD5 hash of the six-character code, rather than the code itself. Since the six-character code is never sent in plaintext through the network, it can be used as a shared secret between the browser and the device, a fact we leverage later to secure the communication between them.

Second, the App Inventor environment in the browser and the Companion App both use AJAX calls to communicate with the rendezvous server. Normally, JavaScript's Same Origin Policy prevents AJAX communication with a server that is not the one from which the JavaScript code being executed was downloaded. This restriction can be mitigated by using appropriate Cross Origin Resource Setting (CORS) declarations on the rendezvous server.

4.2 Two-way communication

Once the handshake with the rendezvous server is complete, the browser and Android device can communicate directly. Upon connection, the browser will send the application's YAIL to the Android device using AJAX requests.[3] When the Companion receives each AJAX request, it evaluates the YAIL and returns an indication of success or failure to the browser.

Since much processing in the Companion occurs asynchronously, errors may occur after an AJAX request is processed. There are also situations where the Companion needs to send unsolicited messages to the browser. To facilitate this, the browser makes separate AJAX requests for any values or other information that the Companion may have for the browser.

The Companion maintains an internal queue of requests to the browser. When a AJAX request for information is sent by the browser to the Companion, it will supply all pending requests as its response. We call the AJAX request that supplies YAIL code to be evaluated a `newblocks` request and the AJAX request to receive returned values a `values` request.

There are two other details governing the AJAX-based communication between the browser and the device. First, we would like to prevent one person from "hacking" another by sending malicious YAIL code to the other person's device. Each `newblocks` request from the browser to the device includes a Message Authentication Code (MAC) calculated based on (1) the YAIL expression, (2) the block ID it is associated with (or -1 if there is none), (3) a sequence number maintained by the browser, and (4) the six-character connection code. Because the six-character code is a shared secret between the browser and device, an attacker cannot create the MAC. However, Companion can verify the MAC, and will only evaluate YAIL code whose MAC it verifies.

[3] The Companion App also provides the appropriate CORS headers so the browser knows it can send AJAX request to the Companion even though the JavaScript running in the browser was not loaded originally from the device.

Second, an AJAX `values` request used by the browser to request information from the device times out after ten seconds and is reissued. So at all times there is a request outstanding from the browser to the Companion for any information it may have. Having at least one AJAX request between the browser and device every ten seconds addresses a practical problem. Many peer-to-peer connections will be on a local network governed by a router with Network Address Translation (NAT) active. Such routers will often drop entries in their address translation tables if there is no communication between devices. Sending a request at least once every ten seconds tends to keep these connections alive.

4.3 Using the USB Cable and/or the Android Emulator

Although we encourage App Inventor programmers to use a WiFi connection when engaging in live development, doing so is not always possible. They may not have a usable local network, or their local network may not support peer-to-peer networking. In these cases, a USB cable can be used to connect the personal computer (PC) running the App Inventor environment to an Android device.

In some cases, the App Inventor programmer may not have access to an Android device. In this case, they can use the Android emulator, which is part of the Google supported Android Software Development Kit (SDK).

Using the USB cable or emulator is architecturally very similar. In both cases we use the `adb` command in the Android SDK to communicate between the PC and the Android device. This command is used to perform various Android debugging tasks. In our case we use it to create a virtual network between the PC and the USB cable or emulator. In particular we instruct `adb` to set up a forwarding of Internet connections to port 8001 on the PC to port 8001 on the connected device or emulator. USB and Emulator connections do not require the rendezvous server, as the device will always be at IP address 127.0.0.1 (the Internet standard "loopback" address that connects a computer to itself).

The final problem is getting the PC's web browser to be able to execute commands like `adb`. This is done by launching an `aistarter` application on the PC that runs a little web server that awaits instructions from the browser and executes them on the PC. The application has full access to PC commands like `adb`.

The MIT App Inventor team distributes a package of setup tools for the Windows, Macintosh, and Linux platforms. These setup tools include the necessary parts of the Android SDK needed for the USB and emulator connections, as well as the `aistarter` application. This setup package is not required for a WiFi connection.

5. Related Work and Discussion

5.1 Influences on our Design

Born out of a collaboration between MIT and Google, and inspired by a long tradition of constructionism-based systems at MIT, App Inventor shares a number of characteristics with other systems created at the institute, such as Logo [21] and Scratch [16]. It also draws from external influences, such as Storytelling Alice [14] and Google's Simple [7]. App Inventor was envisioned as a transformative tool that could turn passive consumers of technology into creators of their own inventions. The work of Caitlin Kelleher in Storytelling Alice [14] reinforced the vision that certain populations, generally underrepresented in the computer science field, could be attracted to the more social uses that mobile devices provide.

Live development as a main characteristic of the system has been described as a story of serendipitous engineering [8]. While Scheme and its REPL capability was first selected as a tool to help with development of the system itself, mostly for ease of debugging and the possibility of incremental development, it turned out to be, not with some additional hard work, the core of live development as

it is available in the system now. This would not have been possible without previous experience with REPL-centric environments that utilize a REPL to interact with running programs, such as Smalltalk [9] (which inspired *DoIt*), Lisp, and Emacs.

5.2 Mobile App Development Frameworks

Most app development frameworks are targeted to experienced programmers. There are a number of mainstream mobile platforms and each of them provide their own tooling, generally around a particular programming language. Android [10] programming is mostly done in Java, on an IDE such as Android Studio or Eclipse; iOS [2] apps can be written in ObjectiveC or Swift, using an IDE provided by Apple; Hybrid and web apps are generally written in JavaScript and other web standard languages. There are a number of efforts to move away from the tooling and restrictions of languages provided by the main platforms. Among others, the Xamarin framework [30] can produce multi-platform apps from a single codebase written in C#, and the RubyMotion project [12] can create native iOS apps written in Ruby.

All of these solutions are very much rooted in the traditional edit-compile-run cycle; the concept of liveness is not present and new testing cycles will reset the app to its initial default state. A number of these systems, though, start showing similarities to some of the basic capabilities that power live development in App Inventor as described in the previous section. Live modifications on apps can be remotely made with systems like RubyMotions [13] or languages like Clojure [22].

In the case of web standard powered apps, development tools such as the Chrome developer tools [11] can also provide some level of interaction with the application running live, by adding breakpoints and even being able to modify certain parts of the code that the browser can automatically re-run. All of these solutions are very barebones, have no visual interfaces, and are targeted to experienced software developers.

Systems such as TouchDevelop [27] provide an interface that is more visual (even providing a 'run' button that can get a script started), but running an app still requires compilation and execution of the new code. A similar example is basic4Android [1], which also simplifies many tasks related to the programming of an app through its visual UI, but coding of the app is textual and the full recompilation cycle is needed.

The closest case to live development as it happens in App Inventor is Mozilla Appmaker [19], a system based on the *web components* emerging web standard with a target audience of non-programmers. The programming models that both systems provide to their users are very different. Appmaker apps are developed in a web browser, and they are created from *bricks* that provide a certain functionality and a series of input and output channels of communication. Bricks can be connected to each other through those channels in order to exchange messages. For instance, a button can be connected to a text box to send it a message that changes the contents of the box. This all happens with no edit-compile-run cycle and no state loss. Programming an app in Appmaker is an exercise of connecting predefined inputs to outputs in the available bricks. Although there exists bricks to create more advanced connections between elements in the app (such as input filters, splitters, or transformers), and the system also provides rather high level bricks such as a *chat room*, there is no concept of programming in terms of many of the most basic computer science principles such as loops, functions, or data structures. Appmaker also provides a *preview mode* that is not directly *live* accessible from a device, and the apps that the tool creates at the time of this writing are limited to Firefox OS devices or to installation in browsers such as Firefox for Android.

5.3 Live Programming

Many discussions of live programming involve a single linear dimension measuring liveness (e.g., [6, 25, 26]). But evaluating and comparing the liveness of programming environments requires considering multiple dimensions.

One dimension is the granularity of program change to which the environment responds, and what actions the programmer must take in order for the system to respond to such a change. An interactive interpreter that responds to individual expressions is more live than a system that processes only whole programs. And a system that responds to individual character edits, such as the code canvas used in Khan Academy programming lessons (e.g., [15]), is even more live. App Inventor's granularity is fairly small: any change within the blocks of a top-level declaration causes that declaration to be re-evaluated. *DoIt* and *Watch* can be used on any block for debugging, but *DoIt* requires an explicit action on the part of the programmer in order to cause a response.

Another dimension is how changes are integrated into a running program. What counts as "live" in this context depends significantly on the domain. When the output of the program is a visual artifact like a picture or a GUI, it makes sense to re-execute the whole program to produce the correct output.[4] In other domains, like interactive synthesized music, there's no need to re-generate past music; only new music going forward should be affect by program changes.

App Inventor is an interpreter-based approach to live programming in which the only re-execution is re-installing top-level declarations (event handlers, global variable declarations, procedure declarations) whenever they change. Users experience this as interactive because code changes are observable in subsequent events, some of which require simple interactions with the device (e.g., touching the screen, shaking/tilting the device), but many of which are automatic (e.g., events that fire due to timers, receiving phone calls and texts, and and callbacks initiated by various methods, such as requesting data from a web service).

App Inventor does not support re-execution-based live programming features of other environments. Consider a program that draws a face on an App Inventor canvas, similar to the one in Khan Academy [15]. Simply redefining an App Inventor variable used by the face-drawing program will not automatically cause the face to be redrawn in the canvas. In order to get this kind of behavior, the App Inventor programmer would have to embed the face-drawing code in some sort of event handler, such as a timer, or create their own use interface slider element associated with the variable that redraws the face every time the variable is changed.

However, in the context of mobile phone apps with potentially very complex behaviors, we view this lack of general re-execution-based live programming as a feature, not a bug. In order to program effectively with re-execution-based live programming, a user must have a very good model of what code is re-executed and what is not re-executed. Consider the concrete example of a button whose text label is initially 0 and is incremented by the value of the global variable n (initially 5) each time it is pressed (Fig. 12).

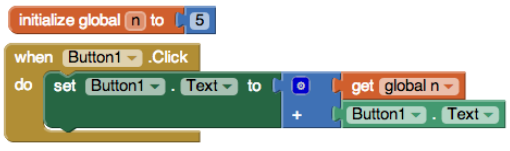


Figure 12. Button click adds the value of `n` to the displayed text.

[4] Sometimes only certain parts of the program need to be re-executed. For example, TouchDevelop cleverly separates GUI-generating code from other code so that GUI updates can be rendered quickly [5].

If the button is clicked twice, its label text changes first to 5, then to 10. Now suppose we edit the definition of n to be 3. Should anything happen to the current label on the button? In certain re-execution models, the two previous button clicks should be replayed, and the button value would change to 6. But in App Inventor, changing n to 3 does not change the current label on the button. However, if we click the button after this change, the button label will change to 10 + 3 = 13. App Inventor users can have a simple model in which changes to their program will only affect future behavior starting at the current system state, not past behavior. In this context, we say that App Inventor obeys *the principle of least surprise* [23], since it would be surprising for users if changing n were to re-execute the previous clicks of the button. And this is a very simple scenario; reasoning about replay in the context of the huge number of events encountered in a typical mobile app would be very hard indeed.

Some limitations of the current version of App Inventor simplify our approach to live programming. For example, all user interface elements must be manually created and arranged within the Designer; it is not possible to write App Inventor programs that dynamically create and arrange such elements. This makes it easy to show the programmer's changes to the user interface. If we enhance App Inventor to allow the dynamic creation of user interfaces, we will need to reconsider how this interacts with live development.

6. Conclusion and Future Work

Live programming is one of the key attractions of App Inventor. It permits the programmer to actually see their program evolve as they write it. By utilizing an intermediate language that supports both interpretive use and compiled code, we can provide both a live interpreted experience for the programmer and native code performance to the application end-user in the finished product.

In the future, we expect to improve the liveness of the Designer by supporting incremental updates when properties are updated in the Designer, avoiding the full application reload we experience today. We also are looking to enhance the way we use the network to better support hotel and school networks, where it is a challenge to get the browser to communicate with the Android device.

Acknowledgments

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

We thank all the developers of App Inventor, current and past, who have contributed to its design and implementation.

References

[1] Anywhere Software. Basic4Android, `http://www.basic4ppc.com/`, accessed Sep. 2, 2014.

[2] Apple Inc. iOS Development Center, `https://developer.apple.com/devcenter/ios/index.action`, accessed Sep. 2, 2014.

[3] P. Bothner. The Kawa Scheme language. `http://www.gnu.org/software/kawa/`, accessed Sep. 2, 2014.

[4] P. Bothner. Hello world in Scheme for Android, 2008. `http://per.bothner.com/blog/2008/AndroidHelloScheme/`, accessed Sep. 2, 2014.

[5] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's alive! continuous feedback in ui programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 95–104, 2013.

[6] M. M. Burnett, J. W. Atwood Jr, and Z. T. Welch. Implementing level 4 liveness in declarative visual programming languages. In *Proceedings of the IEEE Symposium on Visual Languages*, VL '98, pages 126–134, 1998.

[7] H. Czymontek. Simple, `http://google-opensource.blogspot.com/2009/07/programming-made-simple.html`, accessed Sep. 2, 2014.

[8] M. Friedman. The creation of live programming in App Inventor., Aug. 2014. `http://furious-ideas.blogspot.com/2014/08/the-creation-of-live-programming-in-app.html`, accessed Sep. 2, 2014.

[9] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. 1983.

[10] Google Inc., . Android Developers, `http://developer.android.com/index.html`, accessed Sep. 2, 2014.

[11] Google Inc., . Chrome developer tools, `https://developer.chrome.com/devtools`, accessed Sep. 2, 2014.

[12] HipByte.com, . RubyMotion project, `http://www.rubymotion.com/`, accessed Sep. 2, 2014.

[13] HipByte.com, . RubyMotion remote REPL example, `https://www.youtube.com/watch?v=rejYKzLglSE`, accessed Sep. 2, 2014.

[14] C. Kelleher. Storytelling Alice, `http://www.alice.org/kelleher/storytelling/`, accessed Sep. 2, 2014.

[15] Khan Academy. Introduction to Variables, `https://www.khanacademy.org/computing/cs/programming/variables/p/intro-to-variables`, accessed Sep. 2, 2014.

[16] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*, 10(4):1–15, 2010. ISSN 19466226. URL `http://portal.acm.org/citation.cfm?id=1868363\&CFID=113276735\&CFTOKEN=59812236`.

[17] MIT Center for Mobile Learning. App Inventor website, `http://appinventor.mit.edu`, accessed Sep. 2, 2014.

[18] MIT Center for Mobile Learning. MIT App Inventor Public Open Source, `https://github.com/mit-cml/appinventor-sources`, accessed Sep. 2, 2014.

[19] Mozilla Corp. Mozilla Appmaker, `https://apps.webmaker.org`, accessed Sep. 2, 2014.

[20] J. Okerlund. Improving app inventor debugging support, may 2014. Wellesley College undergraduate senior honors thesis.

[21] S. Papert. *Mindstorm: Children, Computers, and Powerful Ideas*. Basic Books, 1980.

[22] W. Peng. Clojure TBNL remote REPL example, `https://www.youtube.com/watch?v=jC-aaIewNkc`, accessed Sep. 2, 2014.

[23] J. H. Saltzer and F. Kaashoek. *Principles of computer system design: an introduction*. Morgan Kaufmann, 2009.

[24] M. Sperber, R. k. Dybvig, M. Flatt, A. Van straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(S1):1–301, Aug. 2009.

[25] S. Tanimoto. A perspective on the evolution of live programming. In *Live Programming (LIVE), 2013 1st International Workshop on*, pages 31–34, May 2013. .

[26] S. L. Tanimoto. Viva: A visual language for image processing. *J. Vis. Lang. Comput.*, 1(2):127–139, June 1990.

[27] N. Tillmann, M. Moskal, J. de Halleux, M. Fahndrich, and S. Burckhardt. Touchdevelop: App development on mobile devices. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, 2012.

[28] 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.

[29] D. Wolber, H. Abelson, E. Spertus, and L. Looney. *App Inventor: Create your Own Android Apps*. O'Reilly Media, Inc., Apr. 2011.

[30] Xamarin Inc. Xamarin Platform, `http://xamarin.com/`, accessed Sep. 2, 2014.