Improving App Inventor Debugging Support

Johanna Okerlund

Submitted in Partial Fulfillment of the
Prerequisite for Honors in Computer Science

May 2014

Acknowledgements

I would first like to thank my advisor, Lyn Turbak, for his relentless support and encouragement. Lyn is one of the most enthusiastic people I have ever met and gets very excited about everything. And his excitement is infectious. After every meeting with Lyn, he has not only conveyed a great amount of knowledge, but has gotten me excited as well.

I'd like to acknowledge the App Inventor development team at MIT, especially Jeff Schiller. Jeff and the team answer questions tirelessly and their enthusiasm for App Inventor never runs out.

Thank you to Stella Kakavouli, Eni Mustafaraj, and Robbie Berg for taking genuine interest in my work and for providing insightful and thoughtful feedback.

Thanks to my family for their investment in everything I do. Few are as lucky as I am to have people who take such an invested interest. And thanks to my little brother for doing what he does.

I would also like to thank the Whiptails for everything. I can't even describe how important the team has been during my time here. I thank them all for being the amazing dynamic people they are. They give me something to look forward to every day.

Lastly, thanks to all the professors I've had and the ones I haven't. It is your energy that creates this environment where we are inspired to pursue what interests us.

Abstract

MIT App Inventor 2 (AI2) is a visual environment where programs for Android mobile devices are composed of blocks resembling puzzle pieces. App Inventor lowers barriers for novices by providing visual guidance for understanding programs and by reducing common programming errors, but it does not eliminate errors entirely. Preliminary analysis of the users' runtime errors shows that better debugging tools for AI2 are needed. People often encounter the same error or a series of errors before they find a solution or give up.

I have implemented AI2 features to help programmers pinpoint the source of runtime errors. In live development mode, AI2 users can test on their devices blocks programs written in a web browser on their computer. Previously, often cryptic runtime error messages were displayed in a dialogue box in the browser window. With my changes, more meaningful runtime error messages are displayed on the block causing the error. I have also implemented a *watch* feature that allows users to track values of variables and expressions. My version is an improved version of the *watch* from App Inventor Classic.

I have also implemented the means to collect more meaningful data on users errors. Currently, runtime error reports in live development mode are automatically stored in a cloud database. The reports include only the error message, the time of the error, and the device on which it was generated. I have augmented the error reports to include the current program code and its author. This extra information can be used to better understand who generates errors, why they are generated, and how users try to fix them. This information will support better debugging and, in the future, can be a basis for an intelligent debugging tutor.

Contents

Chapter 1

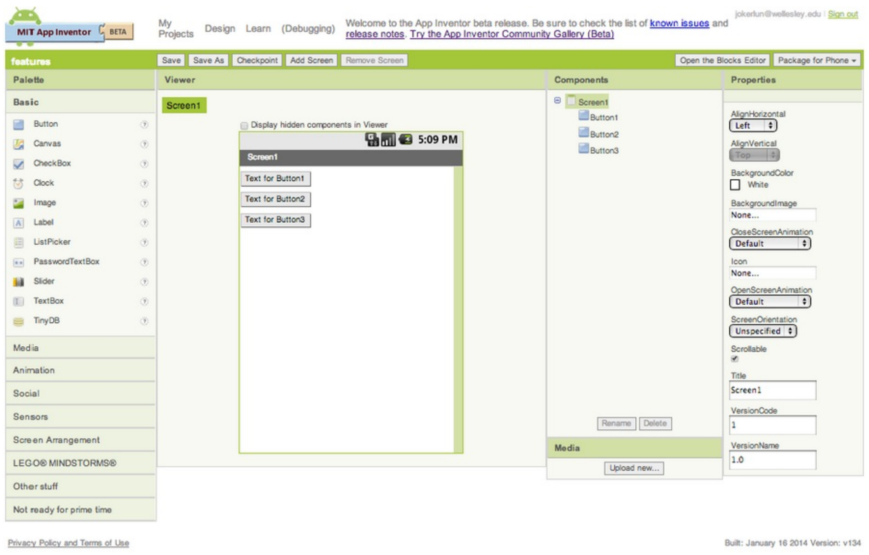
Introduction

App Inventor [Ai1] is an online blocks-based programming environment in which users create apps for mobile Android devices. My work with App Inventor has been to improve debugging support for people using the environment. I have been addressing the needs of people who encounter runtime errors as well as unexpected behaviors in their apps. This work has been in three parts: (1) displaying runtime error messages that (a) are on the block causing the error and (b) are more meaningful than current messages; (2) a *watch* feature that allows users to track how the value of expressions change over time; and (3) setting up the means to collect more data on runtime errors to better understand what additional support users need.

1.1 MIT App Inventor Background

An App Inventor app is specified by a project consisting of a set of user interface components and a program that describes the behavior of these components. The program consists of jigsaw puzzle piece shaped blocks that fit together in ways that are syntactically accurate. Blocks languages like App Inventor lower barriers for novices by providing visual guidance for understanding program constructs and by reducing many common syntax errors.

An App Inventor user programs an app on her computer using the Designer and Blocks Editor. The Designer window is where the user chooses which components to incorporate in her app and arrange how they will be displayed on the phone (Fig. 1-1). Components include buttons, checkboxes, textboxes, animation sprites, any sensors of the phone such as accelerometer and location, and other nonvisible components such as timers and sound file players. Users also choose the initial properties of these components. For example, the user can set the color, text, and size of a button, or the sensitivity of a sensor.



Welcome to the App Inventor beta release. Be sure to check the list of known issues and release notes. Try the App Inventor Community Gallery (Beta)

AlignHorizontal: Left
AlignVertical: Top
BackgroundColor: White
BackgroundImage: None...
CloseScreenAnimation: Default
Icon: None...
OpenScreenAnimation: Default
ScreenOrientation: Unspecified
Scrollable: ✓
Title: Screen1
VersionCode: 1
VersionName: 1.0

Built: January 16 2014 Version: v134

Figure 1-1: The Designer window

The Blocks Editor is where the behavior of the components is specified by a blocks program (Fig. 1-2). Blocks are organized by similar function and stored in drawers located on the left side of the Blocks Editor. Users click the blocks they wish to use and drag them onto the workspace where they fit together with other blocks to make a program.

When a user is programming an app, she has the choice to work in live development mode, where communication between the Blocks Editor and the connected device is constant. The user runs an app on the device called the Companion app. The Companion connects to the Blocks Editor either through WiFi or with a USB cable and runs the application that the user is programming. As the user programs, the Blocks Editor converts new blocks to executable code and sends it to the device, where the code is run. Other communications between the Blocks Editor and the device include requests for values and status updates from the Blocks Editor and responses to these requests as well as error messages from the device (Fig. 1-3).

App Inventor Classic (AI1) refers to an early version of App Inventor. A beta version of App Inventor 2 ([Ai2]) (AI2) was released to power users in July 2013 and to the public in December 2013. Both versions are currently running and available, but App Inventor 2 will soon be the primary version. At the time that my research was conducted, App Inventor Classic had more users and therefore more valuable data to collect, but I implemented new features in App Inventor 2 since that will be more widely used in the future.

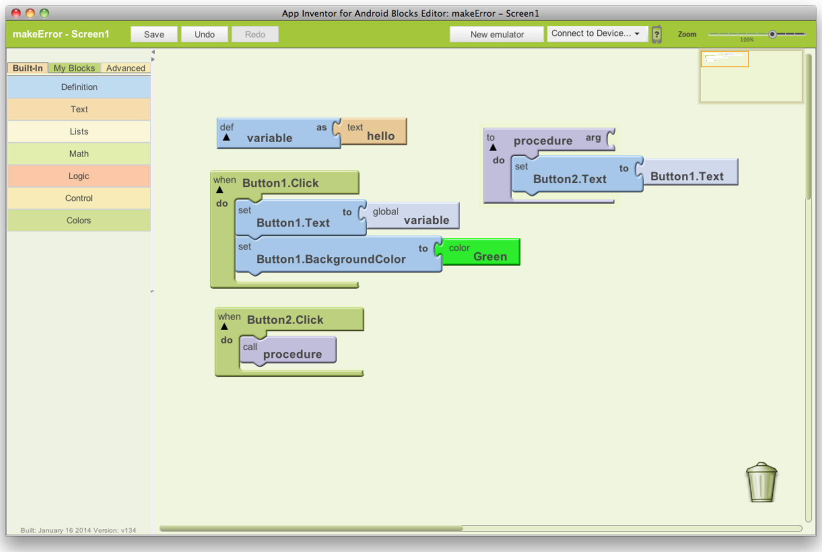

App Inventor for Android Blocks Editor: makeError - Screen1
makeError - Screen1
Save
Undo
Redo
New emulator
Connect to Device...
Zoom
Built-In
My Blocks
Advanced
Definition
Text
Lists
Math
Logic
Control
Colors
def variable as text hello
when Button1.Click
do set Button1.Text to global variable
set Button1.BackgroundColor to color Green
when Button2.Click
do call procedure
to procedure arg
do set Button2.Text to Button1.Text
Built: January 16 2014 Version: v134

Figure 1-2: The blocks editor

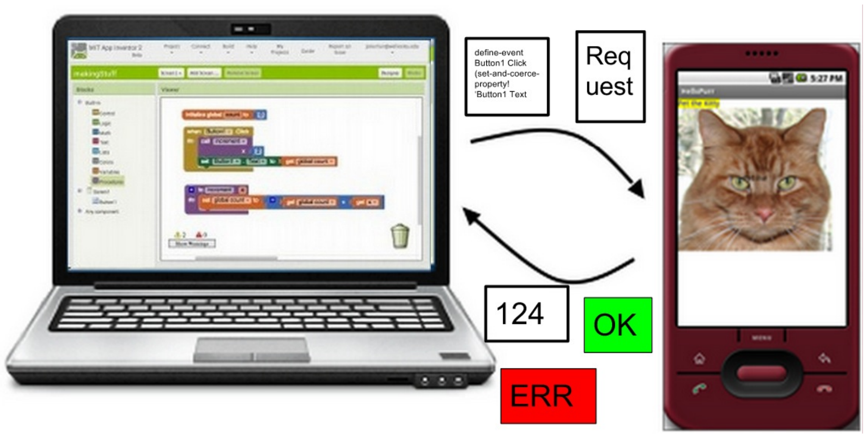

```
define-event
Button1 Click
(set-and-coerce-
property!
'Button1 Text
```

Figure 1-3: Live development mode

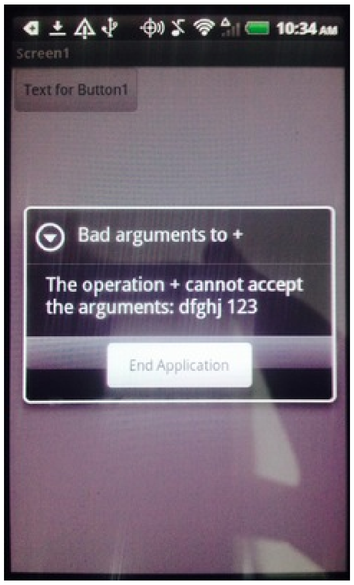
1.2 My Previous Work with App Inventor

Some of my previous work included understanding the structure of the programs people have created. While there was evidence that many people were using App Inventor and a handful of those people were doing very interesting things with it, we had no idea what the rest of the people were doing. The motivation of understanding the structure of programs was to get a sense of who the average App Inventor user is. I collected statistics on the usage of blocks in App Inventor Classic and clustered users based on their use of blocks. This resulted in some conclusions about confusions in the App Inventor interface itself and how people use the environment [OT13].

This work lead to some more questions and a desire to understand users on a deeper level, so I explored errors generated by App Inventor apps (Ch. 4). While the shape of blocks in AI reduces the kinds of errors it is possible to encounter by not allowing users to connect certain blocks, AI does not eliminate runtime errors entirely. Since many people use App Inventor unsupervised and have no prior programming experience, App Inventor errors can be problematic. I collected data on what runtime errors people encounter and how frequently they encounter the same errors. This data led to the conclusion that the debugging support for App Inventor needs to be improved. People generate multiple errors or the same error multiple times. When we stop getting a specific error from a given user, it is because she has either fixed the error or has stopped running the program because she has given up on finding the source of the error. The error messages are vague and provide little instruction about where the user should look to find the source of the error.

1.3 Errors on Blocks

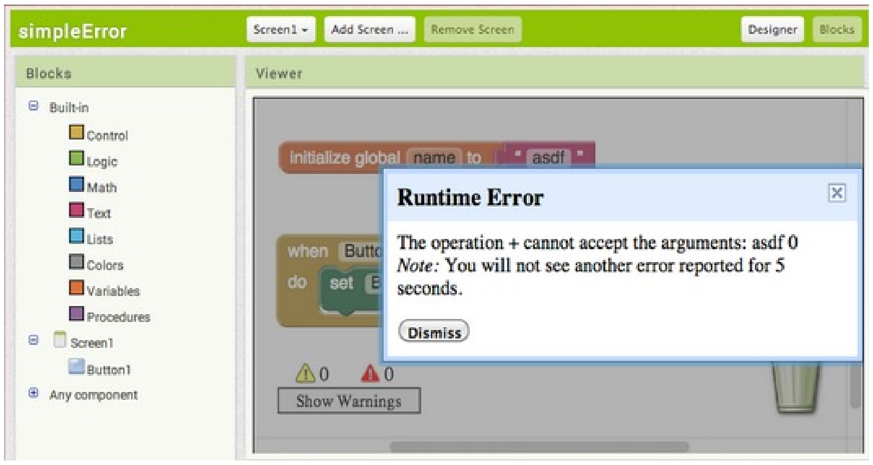
Although blocks eliminate many syntax errors, runtime errors, such as type errors and index-out-of-bounds errors, are still possible. In the current version of App Inventor 2 (version nb132), when a program generates a runtime error, an error message appears both on the device (Fig. 1-4) and in a dismissible dialogue box in the browser (Fig. 1-5). However, this does not give much indication of what generated the error and doesn't allow the user to look at the error and the code at the same time.


Bad arguments to +

The operation + cannot accept the arguments: dfghj 123

Figure 1-4: A runtime error generated by an app displayed on the screen of the device

My solution is to display the error message on the block causing the error (Fig. 1-6). Now the



Runtime Error

The operation + cannot accept the arguments: asdf 0
Note: You will not see another error reported for 5 seconds.

Figure 1-5: A runtime error generated by an app displayed in the Blocks Editor

user has a much clearer point from which to start finding the source of the error. She can also view the error and the code at the same time as she debugs and see multiple errors at the same time (Fig. 1-7). When a block that previously generated an error executes without error, the message is removed from the block. I have also changed the error messages so they are more descriptive, giving users more insight into what exactly was wrong with their code.

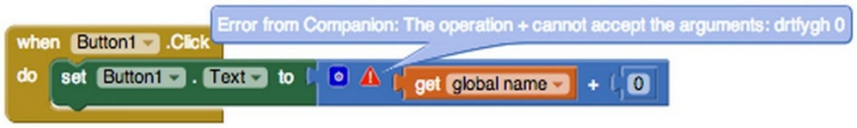
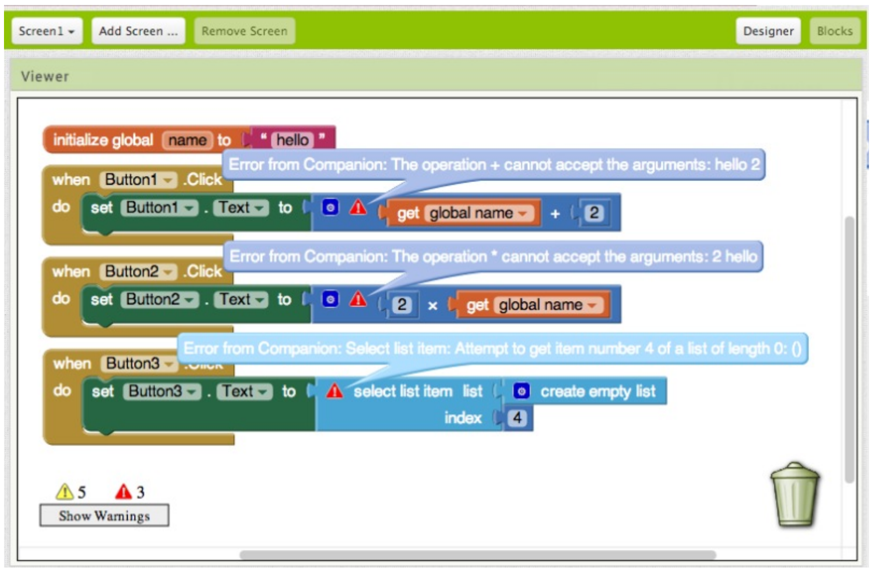

Error from Companion: The operation + cannot accept the arguments: drtfygh 0

Figure 1-6: Error displayed on block

1.4 Watch

As users program in any environment, it is important for them to understand the current state of their program. The state of the program includes information about the code that is running and the history of values of each variable. While the user can use print statements to gain an understanding of the state, it would be better if there were tools to help towards a more complete understanding. App Inventor Classic has a feature called *watch*, which users can use with any expression, including variable references and procedure calls. Placing a *watch* on a block will cause the final value of that expression to be displayed on the block after running the program (Fig. 1-8).

I have implemented a more extensive *watch* feature in App Inventor 2 that allows users to see how values change over time. In my version, instead of just the last value of the expression,



Error from Companion: The operation + cannot accept the arguments: hello 2

Error from Companion: The operation * cannot accept the arguments: 2 hello

Error from Companion: Select list item: Attempt to get item number 4 of a list of length 0: ()

Figure 1-7: Multiple errors displayed at the same time

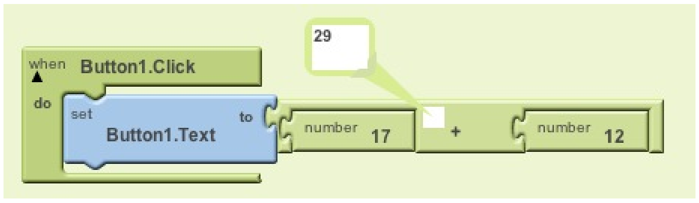


Figure 1-8: Watch in App Inventor Classic

all intermediate values are shown (Fig. 1-9). When a user puts a watch on a block, every time that block is evaluated, the result is displayed in a vertical list, with the most recent value on the bottom. Like in App Inventor Classic, users can watch any expression, including variable references and procedure calls that return values.

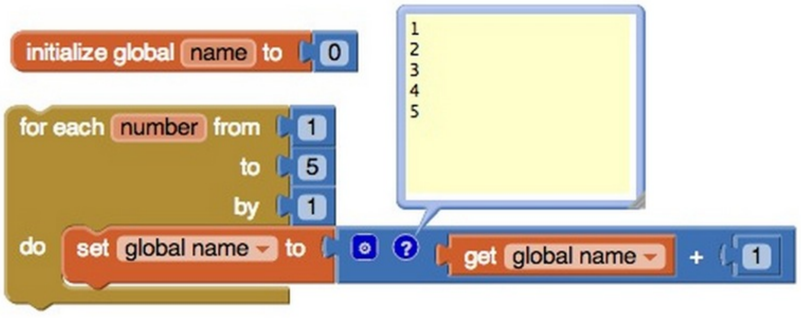
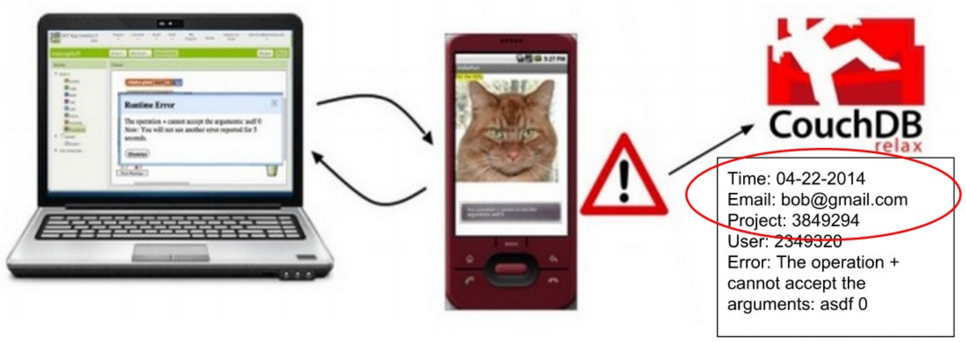


Figure 1-9: The result of watching the addition block during a loop that increments a variable

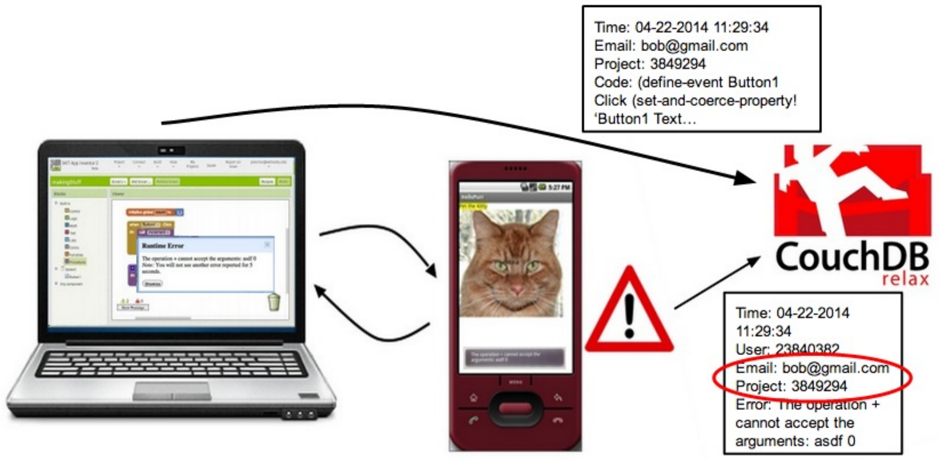
1.5 Data Collection

When a user is in live development mode and a runtime error occurs on the device, the device sends an error report to a database in the cloud (Fig. 1-10). Currently, the helpful information in recorded error reports includes the error message, the time of the error, and a unique ID of the device generating it. Further details about the user and the project are not recorded. I designed the means to trace the errors back to the users and projects generating them. My design is to send the user email and project ID with the error report from the phone. This allows the error reports to be traced back to the user and project that generated them. In addition, the design includes taking a snapshot of the program at the time of the error and sending a report to the database from the Blocks Editor that contains all the current code of the specific project (Fig. 1-11). The two entries in the database would be connected by the user ID, project ID, and timestamp.



Time: 04-22-2014
Email: bob@gmail.com
Project: 3849294
User: 2349320
Error: The operation + cannot accept the arguments: asdf 0

Figure 1-10: An error generated on the device while the user is in live development mode is sent to a database on the cloud



Time: 04-22-2014 11:29:34
Email: bob@gmail.com
Project: 3849294
Code: (define-event Button1 Click (set-and-coerce-property! 'Button1 Text...

Time: 04-22-2014 11:29:34
User: 23840382
Email: bob@gmail.com
Project: 3849294
Error: The operation + cannot accept the arguments: asdf 0

Figure 1-11: When an error is generated, reports are sent from both the Blocks Editor and the device.

1.6 App Inventor Architecture

App Inventor consists of several different parts that are implemented with different languages. Because my implementation was done in App Inventor 2, I will discuss the architecture of the recent version.

The Companion app running on the phone is written in Kawa, a version of Scheme implemented in Java that can access Java features, including the Android SDK. The Companion connects to the Blocks Editor through WiFi or a USB cable and runs the app the user is programming as well as communicates information back to the Blocks Editor.

The Blocks Editor is run in the browser and is implemented with Java and JavaScript. The blocks, the behavior of the blocks, and communication between the Blocks Editor and the device during live development are written in JavaScript. App Inventor 2 blocks are impemented in Blockly, a JavaScript open source blocks language developed by Neil Fraser intended for people to use in any blocks based environments they are developing.

App Inventor is deployed with Google App Engine, a platform where developers can upload and run their applications without having to worry about maintaining a server. App Engine also allows Google usernames to be used for authentication to the deployed app and provides the means for storage of information for each user. For App Inventor, this means the users' projects are stored on the cloud with App Engine. Code in App Inventor that deals with loading and storing projects is written in Java.

For storage, each project is represented by a few text files. One contains an XML representation of the blocks in the Blocks Editor, including where the blocks are located. Another file contains a JSON representation of the components the user has chosen in the Designer and the properties of those components.

App Inventor is deployed for people to use, but it is also open source, which means anyone can obtain a copy of the source code, make improvements, and run their own version. The source code for App Inventor is stored on GitHub, a project storage and sharing service with revision control. For a developer to get the source code on her own computer, she makes a clone of the main version of the source code, the master branch, to her local machine and can make changes locally without affecting the master. When a developer has made a change she wishes to incorporate into the master branch, she submits a pull request and one of the main developers will test and review the code to decide if it's ready to be merged in with the master.

While a developer has a version of the source code on her local machine and she wants to run it, she uses Apache Ant to build and launches App Inventor on a port on localhost. Making changes to the Companion app requires building those files and downloading the new Companion to the device.

1.7 Road Map

The rest of this paper is organized as follows:

- Chapter 2 discusses related work on debugging as it relates to blocks languages.
- Chapter 3 gives details about errors in App Inventor.
- Chapter 4 describes my previous work collecting data on App Inventor errors and conclusions drawn.
- Chapter 5 discusses my design and implementation of the mechanism that displays runtime error messages on the block that generated them.
- Chapter 6 discusses my design and implementation of *watch*.
- Chapter 7 talks in detail about the design and implementation of sending extra information in the error reports.
- Chapter 8 summarizes my work and discusses future work and extensions to my project.
- The appendices contain the relevant code for the implementations discussed in Chapters 5, 6, and 7.

Chapter 2

Related Work

2.1 The State of Modern Debugging

2.1.1 The Debugging Scandal

Much work has been done to study and critique the current tools for debugging support in programming environments and identify properties of successful debugging tools. The April, 1997, issue of the *Communications of the ACM* focused on the state of debugging tools at the time hightlighted the limits of existing tools and coined the term the *Debugging Scandal*. With all the work done to make computers faster, more powerful, and more efficient, debugging tools have the potential to be very powerful [Lie97].

An article from the same publication of CACM discusses how important the principle of *immediacy* is when designing debugging tools. Temporally, this means that if a user has to wait for a result, she loses her train of thought and the debugging process is interrupted. Spacial immediacy is also important since it becomes frustrating if related pieces of information are far away. Semantic immediacy means that visual and textual representations of data closely resemble the data itself. The user should be able to see the representation and immediately know what it means. All of these principles foster a more direct relationship between the user and the program and a more immediate understanding of what is happening during execution. [ULF97]

Baecker, DiGiano, and Marcus discuss key principles of visualizations as debugging tools. A program's state is understood by its data and visualizations can be incredibly effective at expressing state if they are able to clearly represent the data and how it changes over time. Animations that use size, spacing, and color, as well as visual and audio cues can very accurately represent the execution of a piece of code. [BDM97]

My work to implement debugging support for App Inventor draws upon these principles. Placing error messages on the block that caused the error is an example of spacial immediacy [ULF97] since the information is displayed where the user would be fixing the error. My improvements on the error messages themselves is an improvement on semantic immediacy, making the error messages a more accurate representation of what went wrong with the data. My implementation of *watch* is the first step towards a system that allows users to fully understand the state of the data in their program.

2.1.2 Deterministic Replay Debuggers

Some of the most recent debugging research centers around the design and implementation of *deterministic replay debuggers.* These are debuggers that store everything that happens during the execution of a program and present it afterward for the user to view. There is also a way for the user to manipulate time and see how things changed and what else was happening at the time of that change. Implementing such debuggers requires storing the required information in a way that won't affect the execution of the program or slow it down too much as well as displaying the information to the user meaningfully.

Mugshot [MEH10] is a system for capturing the behavior of JavaScript applications. Mugshot runs without using much storage and without slowing execution down by much and allows users full access to everything that happened during execution [MEH10]. WaRR [AC11] and Jalangi [Sen+13] are also systems for web application analysis. RERAN [Gom+13] is a replay system for apps on Android devices. Replay debuggers for GUIs on phones had previously not provided enough information about details of the users' interactions to be particularly helpful. RERAN records a more complete view of the actions performed by the user and the data taken from the phone's sensors to replay the app more accurately. While I am not implementing a deterministic replay debugger, my work with *watch* is the first step towards one for App Inventor. Possible designs are discussed in (Sec. 8.2) and my implementation of augmented code (Sec. 5.3) would be helpful in the implementation of such designs.

2.2 Debugging in Other Blocks Languages

2.2.1 Scratch

Scratch 2.0 [Scr] is a blocks language for programming animations, games, and stories. The blocks control the actions of sprites on a canvas. The programming environment consists of the blocks canvas on the right, the canvas with the sprite(s) on the left, drawers of blocks to choose from in the middle, and a list of all sprites on the bottom left (Fig. 2-1).

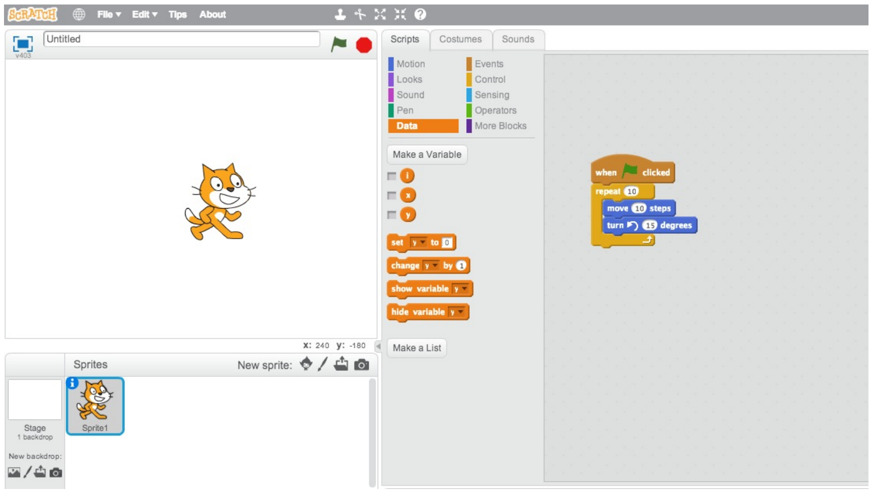


Figure 2-1: The Scratch Designer window

Variables in Scratch are either specific to a sprite or global across the whole project. In the top left of the sprite canvas, all variables are displayed so the user can see them change during the execution of the program (Fig. 2-2). The user can choose which variables to show and which to hide in the variable drawer.

The Scratch language is designed to try to prevent the generation of runtime errors altogether. If a user tries to perform math operations on a string, which would normally generate a runtime error, it will substitute the number "0" for the string so the program runs without crashing. If a program generates a list indexing error, the error-generating line of code is ignored and the rest of the program continues.

For behavioral or logical errors, Scratch has a feature called *single step mode*, where users can step through their code at various speeds. Users can also choose to have code outlined as it executes so they can track when various blocks are called. Scratch 2.0 does not have this feature, but previous versions did.

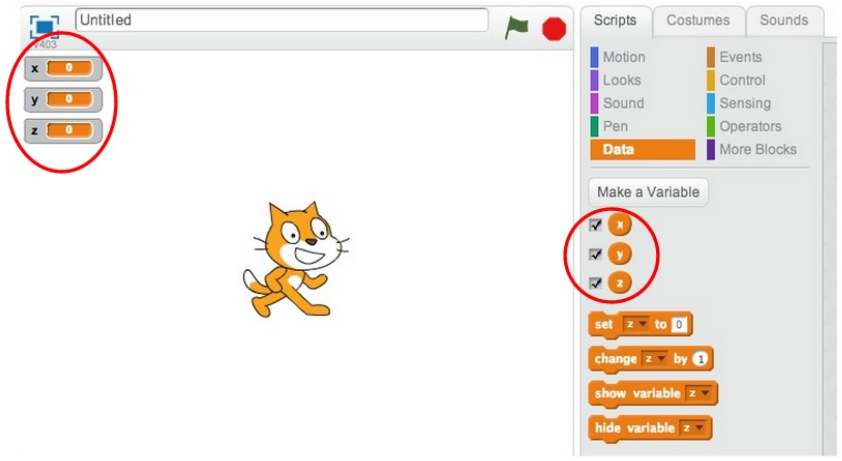


Figure 2-2: Variable watch in Scratch

2.2.2 StarLogo TNG

StarLogo TNG [Sta] is an environment for controlling 3D characters' movement, appearance, and sound with blocks programming (Fig. 2-3). Like Scratch, StarLogo TNG tries to completely eliminate the existence of runtime errors. The language is statically typed, meaning that types are checked during compilation. Block types are represented by shape, and when a variable is declared, all references to that variable change shape as well (Fig. 2-4). This prevents type errors from occurring. With other kinds of errors, such as list indexing, the program does not produce an error; rather, the block causing the error simply doesn't execute and the rest of the program carries on as usual, as it does in Scratch.

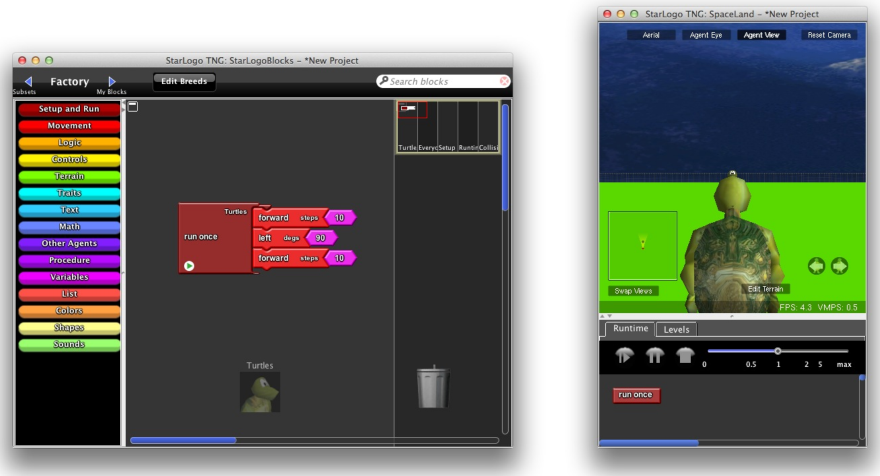


Figure 2-3: StarLogo TNG blocks and 3D canvas

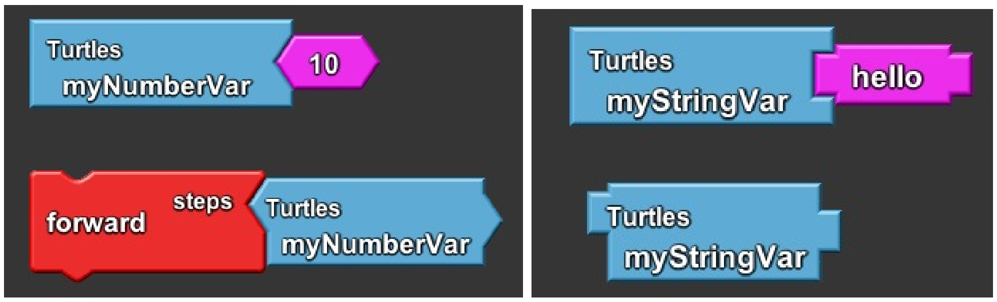


Figure 2-4: StarLogo variable declarations and references of different types

Roque's thesis proposal [Roq06] outlines a design of improved debugging support for StarLogo TNG. The plan is to implement a single stepping mode, similar to the one Scratch has (Sec. 2.2.1). The design includes indicating which blocks are being executed and panels to show the current state of variables. Users will also have the choice to step one block at a time, step into a procedure, or return to exit the procedure they are in [Roq06].

App Inventor does not have a single stepping mode like StarLogo TNG or Scratch, but it would be incredibly helpful for users to be able to better understand the flow of their running programs. My implementation of augmented code (discussed Sec. 5.3) could be a step towards a single step mode for App Inventor.

2.3 Debugging in AI1

2.3.1 Do It

Users can right-click a block and perform a *Do It* action. When selected, the block is immediately compiled and sent to the device, where it is executed. The result is send back to the Blocks Editor where it is displayed on the block. Users can perform *Do Its* on expressions (Fig. 2-5) or on statements (Fig. 2-6). Statements don't return values, so the result displayed on the block is empty. This is a very useful feature because it allows users to isolate pieces of code to execute alone, which helps track errors. It is the first step towards a single stepping mode like Scratch and StarLogo TNG. Both App Inventor Classic and App Inventor 2 have a *Do It* feature.

2.3.2 Watch

App Inventor Classic has a basic implementation of *watch*. Users can put a *watch* on an expression and, after the execution of the program, see the final value of that expression (Fig. 2-7). This is

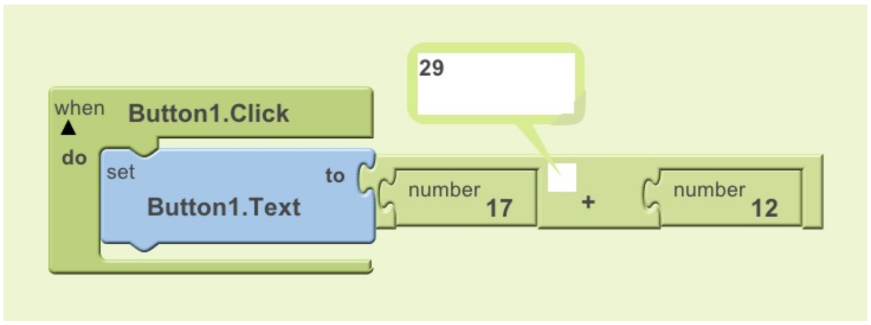


Figure 2-5: A *Do It* on an expression

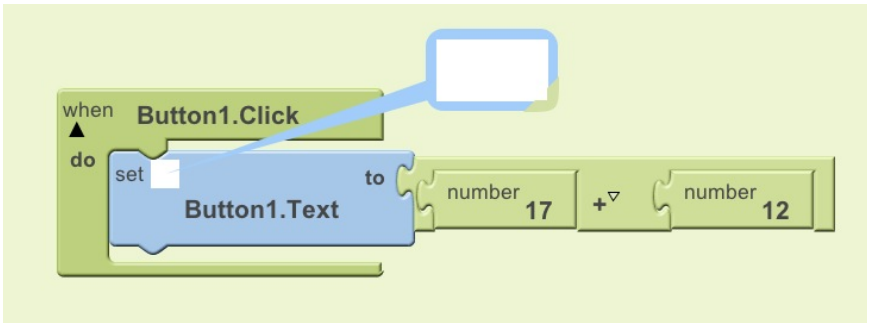


Figure 2-6: A *Do It* on a statement

helpful to an extent, but doesn't show the user how the expression changes during the program execution. If the expression is evaluated multiple times, only the value from the last evaluation is displayed (Fig. 2-8). The current version of App Inventor 2 does not include a *watch* feature, giving the perfect opportunity to improve on its design (Sec. B).

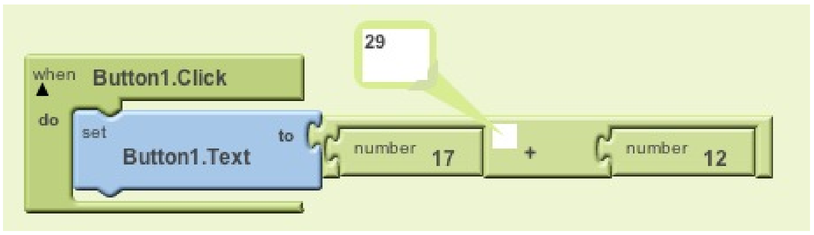


Figure 2-7: Watch in App Inventor Classic

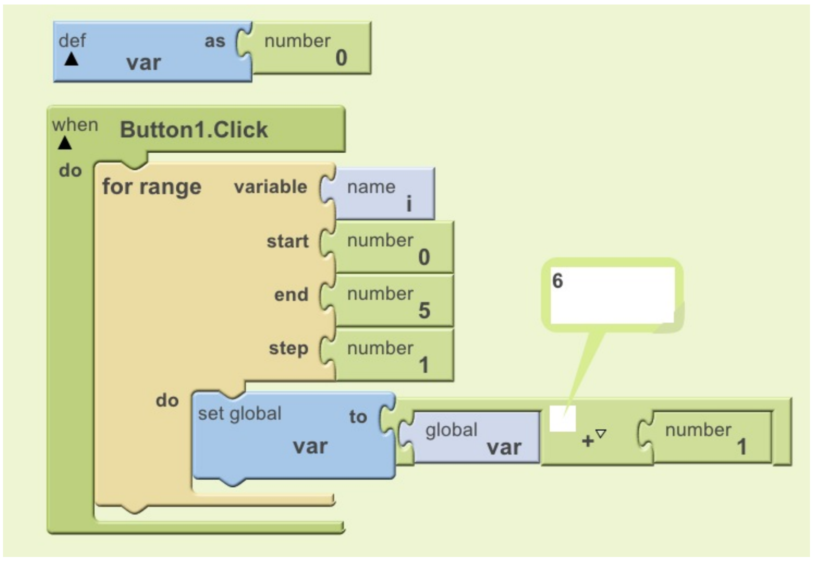


Figure 2-8: A watched expression in a loop

Chapter 3

Errors in App Inventor

3.1 Code-time Errors

Code-time errors occur when the user tries to do something that is not allowed and the Blocks Editor catches the mistake. Shapes of blocks are helpful and reduce the number of code-time errors that can occur. For example, App Inventor does not allow a statement to be given as a parameter to another statement. However, there are certain distinctions, such as type, that are not represented by different blocks. The types of value blocks in App Inventor are number, string, boolean, and color, all of which have the same connector shape (Fig. 3-1).

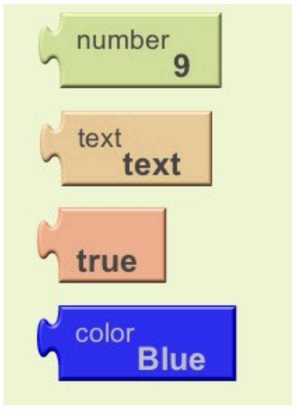
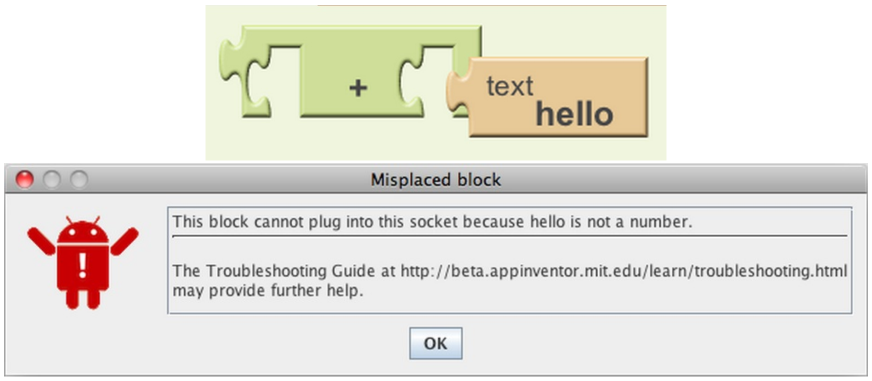


Figure 3-1: Example of number, string, boolean, and color blocks

Since App Inventor block shapes do not distinguish type, the shapes of the blocks imply that an expression block of any type is syntactically allowed to go anywhere. This is not true, however, and the App Inventor Blocks Editor alerts the user if she tries to do certain things that are not allowed. For example, if she tries to plug a string into a math operator, the Blocks Editor prevents the connection and moves the block with the plug away from the socket, giving it the appearance of jumping out of the socket. In App Inventor Classic, a warning message appears (Fig. 3-2), but in App Inventor 2, the block jumps out without an explanation.

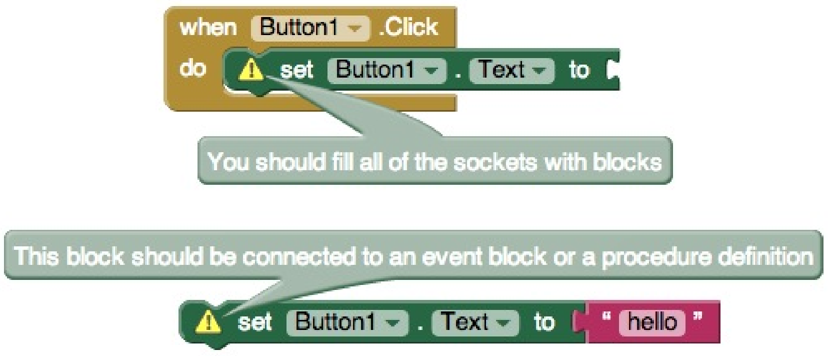
App Inventor 2 introduces error and warning triangles that go on blocks. Warnings let the user know of what might cause a problem, but not necessarily errors during execution. For example, warnings alert the user to unfilled sockets and code that will not be executed (Fig. 3-3). Because the user has blocks with unfilled sockets while she programs, all warnings can be toggled on and off so she doesn't get annoyed.



This block cannot plug into this socket because hello is not a number.

The Troubleshooting Guide at http://beta.appinventor.mit.edu/learn/troubleshooting.html may provide further help.

Figure 3-2: If a user tries to plug a block of the wrong type into a socket in AI Classic, an error message appears.



You should fill all of the sockets with blocks

This block should be connected to an event block or a procedure definition

Figure 3-3: Warnings appear on blocks in AI2.

Error triangles indicate more serious problems, such as when a variable is out of scope (Fig. 3-4) or when there is a network connection error. If there is an error triangle on a block, this block is not compiled and the relevant code is sent to the device until the user has fixed the error.

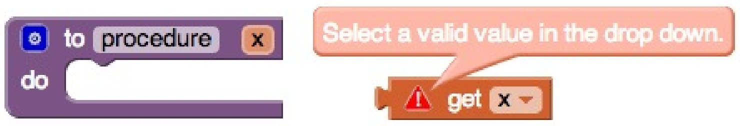


Figure 3-4: Error triangle in AI2 for out of scope variable

3.2 Logical Errors

Logical errors happen when a program behaves differently from how the user expects it to behave. A value might be different from what the user expected it to be or certain events might be happening when they aren't expected to. Programmers use general techniques such as breakpoints and print statements to pinpoint the source of the error. It is hard for the programming environment to provide direct help to the user because it does not know what the user's intention is. However, the environment can provide tools to help the user understand the state of the variables in the program and what happens during execution. App Inventor Classic has two features that do this (Sec. 2.3): *Do It* (Sec. 2.3.1) and *watch* (Sec. 2.3.2). App Inventor 2 has *Do It*, but does not have a *watch* feature yet (discussed in Sec. B).

3.3 Runtime Errors

Runtime errors are errors that are not caught by the Blocks Editor, but that are generated during execution of the program. App Inventor does not check the types of all blocks in the Blocks Editor. Thus, if a variable is an illegal type, it will cause a runtime error even though the Blocks Editor allowed it (Fig. 3-5). This is an example of a **Dynamic Type Error**.

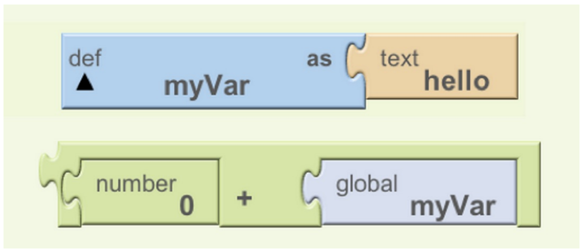


Figure 3-5: Error: The operation + cannot accept the arguments 0 hello

Dynamic Value Errors are another type of runtime error and occur when the programmer uses a value incorrectly. An example of a dynamic value error is list index out of bounds errors (Fig. 3-6).

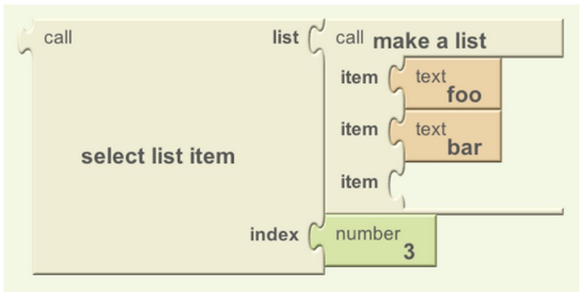


Figure 3-6: Error: Select list item: Attempt to get list item number 3 of a list of length 2: (foo bar)

Syntax Errors can also occur in App Inventor. While blocks languages try to physically prevent all syntax errors, some may still be possible. App Inventor Classic uses `name` and `value` blocks to represent variables. The visual appearance of these blocks is similar (Fig. 3-7) and the difference in how they are used is subtle to a novice programmer. Name blocks are used to declare parameters to functions and event handlers. The value blocks are used in the body of the function or event handler to refer to the value of the variable declared by the name block.

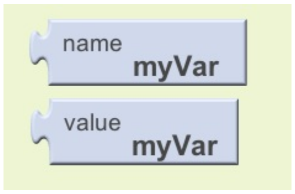


Figure 3-7: The difference between name and value blocks is subtle.

Because of the confusion between name and value blocks, users sometimes unplug the name parameters from event handlers and try to use them in the body of the event (Fig. 3-8). They receive a cryptic error message that the event has too many arguments. This is because App Inventor successfully compiles code in the case where a required variable declaration socket is left empty. This should be caught at code time, but for some reason is not.

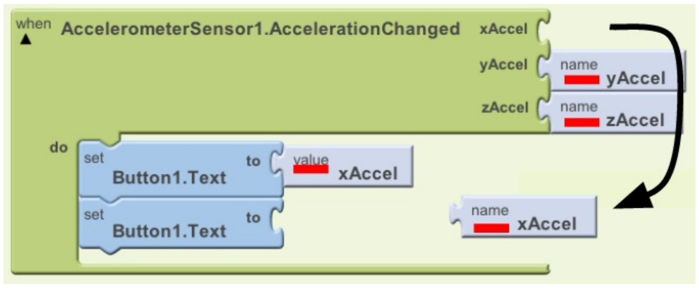


Figure 3-8: Error: Call to 'AccelerometerSensor1 AccelerationChanged' has too many arguments

Chapter 4

My Previous Work Analyzing AI1 Errors

My previous work with App Inventor related to this project was to understand the types of errors users get and how they try to fix them. When a user is in live development mode and a runtime error occurs on the device, the device sends an error report to a CouchDB database in the cloud via ACRA (Application Crash Report for Android). I was given access to this database and performed preliminary analysis on the types and frequencies of App Inventor Errors.

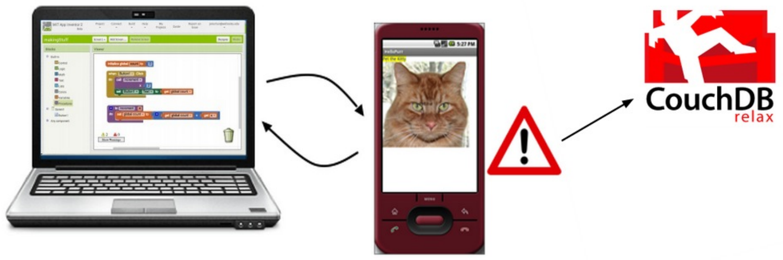


Figure 4-1: An error generated on the device is to a database.

The error reports I analyzed were the errors generated by 2,314 people during a 2.5 week period in July 2013 and are all from App Inventor Classic. The most common type of error is a type error, followed by the syntax error (Fig. 3-8) that is a result of unplugging name declaration blocks (Fig. 4-2). The specifics of some of these errors are discussed in Sec. 3.3.

An analysis was also done of the average number of errors per user per day (Fig. 4-3). This data was collected at a time when every error that occurred on the device was sent to the database regardless of how many times it had already been sent. *Quick-fire events* are events that potentially

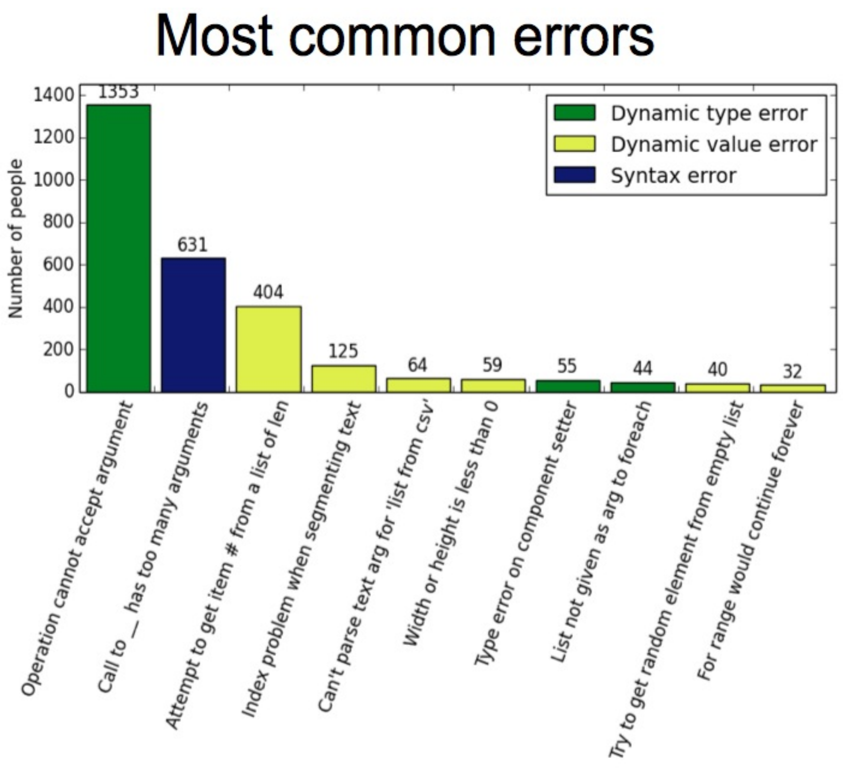


Figure 4-2: The most common errors in App Inventor Classic

fire many times per second such as acceleration changed, orientation changed, and clocks. This means that if a quick-fire event generates an error, it does so hundreds or thousands of times and they are all recorded in the database. This is why some people have thousands of errors; they might just have one event firing one error many times before they can stop the execution of the program. The users in the lower third, who only get a few errors per day, are able to fix their errors, or give up. The ones who fix the error are not of concern, but we never know if some of the people give up on App Inventor after encountering the error a few times without being able to figure it out. The people in the middle range are the ones that are of real concern. Users who get more than 10 errors, but less than about 100, per day aren't getting enough for them to be quick-fire errors, but getting enough to show that they need help finding the source of the errors. About a quarter of the 2,314 users fall into this category.

The last piece of information gathered from the error reports was how many times a user gets the same error (Fig. 4-4). This count does not include the quick fire errors; it only includes errors that users get by rerunning the program themselves. When an error occurs and the error message appears on the device, the only option is to quit the app. The user can press the back button on the phone and continue running the app, but not many people know about this. The error reports

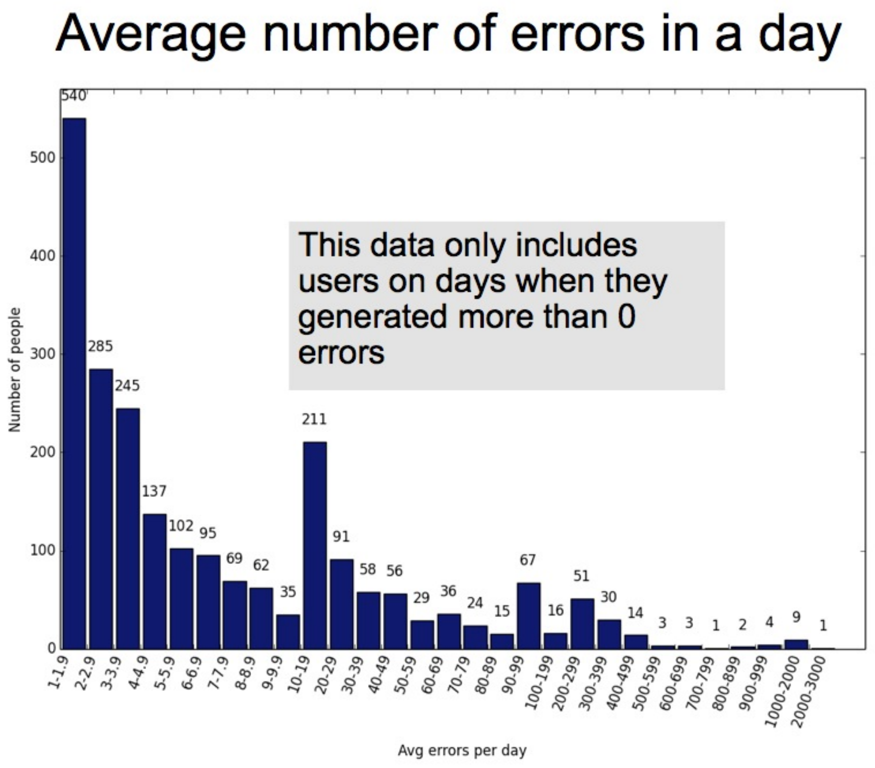


Figure 4-3: Average Errors Per Day

have a field for the time the app was started at, so if two error reports by the same user have different start dates, this means that the user closed the app between the time the two reports were generated and so we know that they were not generated by a quick-fire event. The data collected from counting repeated errors shows that people can't figure out the source of their error or make the same mistake in the program repeatedly. Somewhere, there exists a gap in understanding. It is unknown whether these people eventually fix their error or give up on App Inventor. However, this data shows that these people need help debugging their App Inventor apps.

The data reported in this section was collected from App Inventor Classic because, at the time, it was the version most widely used and had the most data. However, because of the switch from App Inventor Classic to App Inventor 2, my implementations discussed in Sec. A, Sec. B, and Sec. 7 are all done in App Inventor 2.

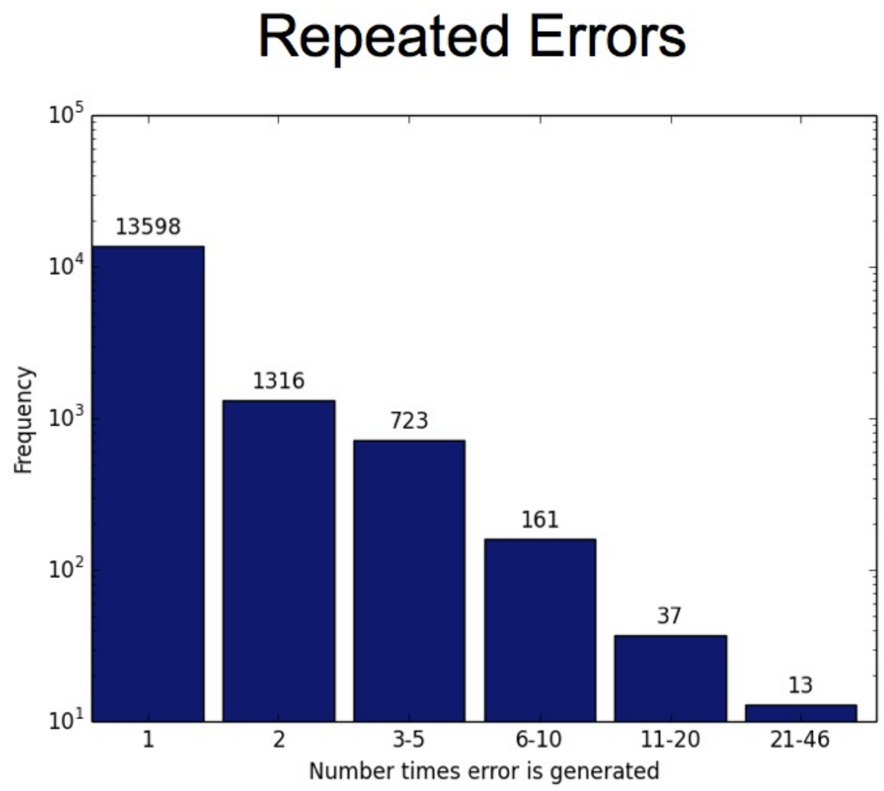


Figure 4-4: How many times users get the same errors (does not include quick-fire errors)

Chapter 5

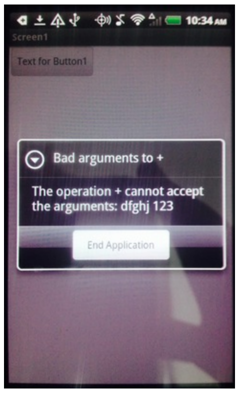
Error Messages On Blocks

This chapter focuses on my work displaying error messages on the blocks that generated them. The implementation first required understanding the communication between the Blocks Editor and the connected device and what happens on both sides when a runtime error occurs.

5.1 When an Error Occurs

App Inventor Classic

In AI Classic, when a user is programming in live development mode and her program generates a runtime error, a message appears on the phone with a single choice to end the application (Fig. 5-1). The only way the user can get back to the application without reconnecting the device to the Blocks Editor is by pressing the back button on the phone, but this not obvious and not many users are aware of this feature.



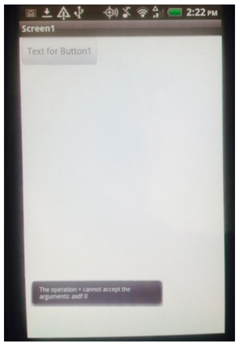
Bad arguments to +

The operation + cannot accept the arguments: dfghj 123

Figure 5-1: In App Inventor Classic, a running app generates an error and a message appears on the screen of the device.

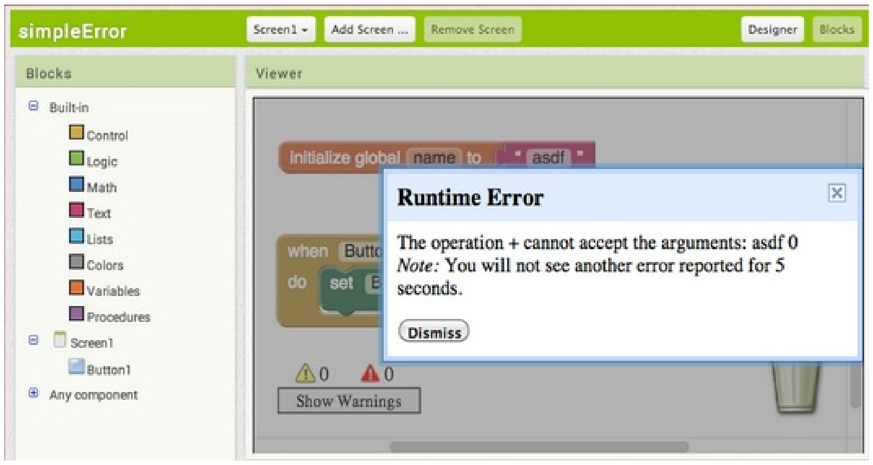
App Inventor 2

In App Inventor 2, a message appears on both the device and the Blocks Editor. The message on the device stays visible for only a few seconds. Its purpose is to direct the programmer's attention to the Blocks Editor, where the user has the single option to dismiss it. This is an improvement over App Inventor Classic because it indicates that something in the blocks program is wrong and the programmer needs to fix it. However, the issues remain that the message does not give much insight into where the error is located, users can't view their code and the error message at the same time, and errors caused by quick-fire events still get in the way of the user coding.



The operation + cannot accept the arguments: asdf 0

Figure 5-2: Error message from App Inventor 2



Runtime Error

The operation + cannot accept the arguments: asdf 0
Note: You will not see another error reported for 5 seconds.

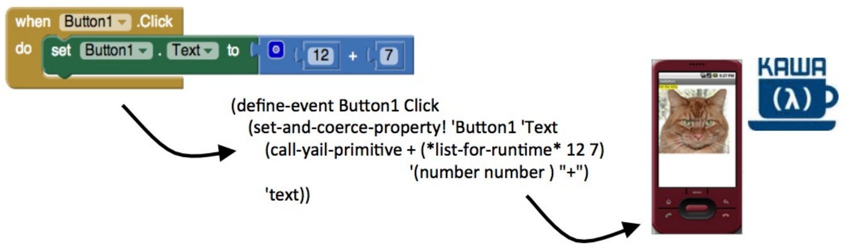
Dismiss

Figure 5-3: The error message is displayed in the Blocks Editor in App Inventor 2

5.2 Communication between the Blocks Editor and the Android Device

Any improvement in debugging support first requires understanding the communication between the Blocks Editor, where the program is written, and the device where the program is executed and the error occurs. The device has to successfully communicate relevant information to the Blocks Editor about the error that will help the user pinpoint its source. The Blocks Editor needs to be able to express the information about the error to the user as clearly as possible.

5.2.1 From the Blocks Editor to the Device



```
(define-event Button1 Click
  (set-and-coerce-property! 'Button1 'Text
    (call-yail-primitive + (*list-for-runtime* 12 7)
                           '(number number ) "+")
    'text))
```

Figure 5-4: Blocks compiled and sent to the device

In App Inventor 2, without my changes, the blocks of the program are converted into YAIL (Young Android Intermediate Language), a language that is very similar to Kawa, and sent to the device (Fig. 5-4). Kawa is a version of Scheme implemented in Java, and in AI2 has access to all Android libraries. The device has an application on it called the Companion. When the user is in live development mode, she runs the Companion app on the device to connect to the Blocks Editor and run the current version of the code.

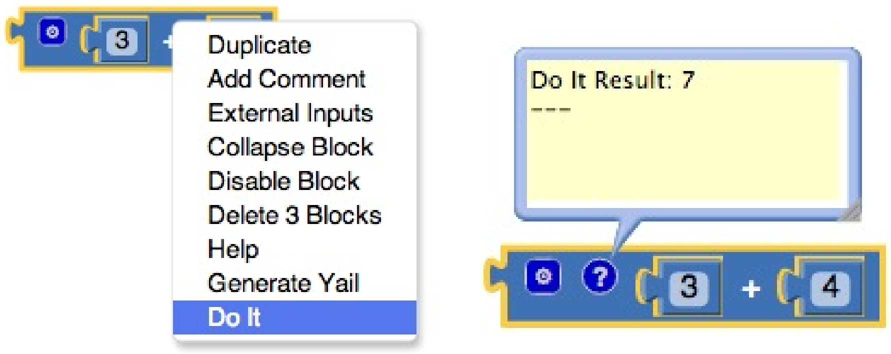
The Companion runs a web server that the Blocks Editor can communicate with via HTTP requests. These requests allow for an ongoing two-way connection between the Blocks Editor and the device. The Companion is also running a YAIL interpreter written in Kawa that executes the code that comes to the device from the Blocks Editor as part of the HTTP requests. The Blocks Editor sends several types of information and requests to the device:

- Code to be executed: When new code is sent to the device, the device interprets the code.
- Request to perform a *Do it*: In the Blocks Editor in AI Classic and AI2, the user has the option to request that a block is executed immediately. The response to this request is the result of the code that was evaluated.

- Requests for information: The Blocks Editor periodically sends requests to the device to make sure nothing has gone wrong. The device either responds with *OK* or an error message. This simulates a TCP-like connection. Sometimes the device has information to send to the Blocks Editor, but it cannot send it unless it is responding to a request. Therefore, the Blocks Editor continually sends requests for information.

Each time the user changes or adds code in the Blocks Editor, all the code of the program is recompiled. A hash map keeps track of the code for each of the top level blocks. If any of the code within a top level block has been changed, the entire block of code is sent over to the device. Thus, while the code for the entire program is recompiled to YAIL, only code that has been changed is sent to the device. When the device receives new code, it updates the version of the user's app running on the phone. Events are remembered in the event registry and variables are stored in an environment.

The Blocks Editor can also send requests for information to the device. App Inventor has a feature called *Do it*, which can be accessed by right clicking on a block (Fig. 5-5).



Do It Result: 7

Figure 5-5: Do It

The code from the selected block is sent to the phone to be executed. The result is returned to the blocks editor and displayed in a comment on the block (Fig. 5-5).

5.2.2 From the Device to the Blocks Editor

The device puts together packets of information called *retvals* (return values). The retvals are sent back to the Blocks Editor to be processed. *Retvals* are JSON strings that contain fields to indicate the type of message and any other relevant information.

There are three different types of retvals:

- *return*: Every time a piece of code is sent to the phone, the phone sends a return message to the Blocks Editor with a status of *NOK* or *OK* depending on whether there was a problem or not. In particular, it returns *NOK* if there is a network connection error or if the code the device received has empty sockets. When the Blocks Editor receives a return with anything other than *OK*, it puts a warning on the top level block that the code corresponds to.

  The return retval also has the option of an included value field. If the user selects *Do It* on a block, the code is sent immediately to the Blocks Editor with the block ID to be executed. The result of the execution is sent back to the Blocks Editor in the value field of the return retval and the result of the *Do It* is placed on the correct block in a comment bubble (Fig. 5-5).

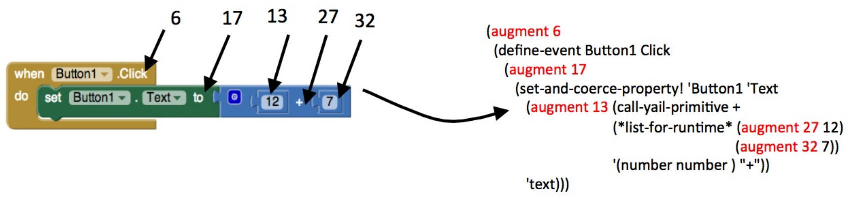
- *error*: If a runtime error occurs while the app is running, a retval of type error sends the error message back to the Blocks Editor.

- change screens: If the user changes screens on the app on her phone, a change screens retval is sent to the Blocks Editor so it can redisplay the blocks of the correct screen.

5.3 My solution: Errors on Blocks

5.3.1 Augmented Code

The previous mechanism for converting blocks to YAIL did not include any information about block identities for general code updates, so there was no way to associate errors with particular blocks. When the code was evaluated, neither the device nor the Blocks Editor knew which block was being executed. Blocks in an App Inventor have a unique ID for a given app screen. With my implementation (Appendix A), when the blocks are converted to YAIL, the code is wrapped in a tag that has the word `augment` and the block ID (Fig. 5-6). When the interpreter evaluates the YAIL code, it keeps track of the block ID for the current expression in such a way that if an error occurs within that expression, it can associate the error with the block ID.



```
(augment 6
 (define-event Button1 Click
   (augment 17
     (set-and-coerce-property! 'Button1 'Text
       (augment 13 (call-yail-primitive +
                          (*list-for-runtime* (augment 27 12)
                                              (augment 32 7))
                          '(number number ) "+"))
       'text)))
```

Figure 5-6: Blocks compiled to augmented YAIL and sent to device

With my changes, when an error occurs, because of the augmented code, the device knows which block generated the error and sends its ID back to the Blocks Editor when it sends the error message. When the Blocks Editor receives an error message and a block ID, it puts the error on the correct block. If that block is collapsed or is in a group of blocks that is collapsed, it expands the block and all blocks in the group group so the user sees the message. It does this by recursively walking down the tree of blocks.

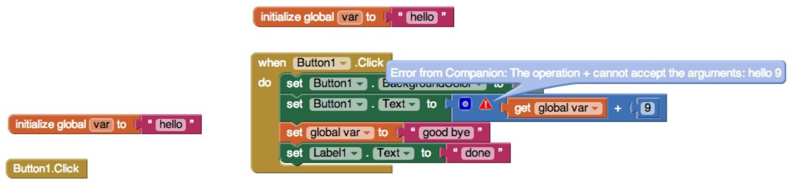

Error from Companion: The operation + cannot accept the arguments: hello 9
initialize global var to "hello"
when Button1 .Click
do
set Button1 . BackgroundColor to
set Button1 . Text to
get global var + 9
set global var to "good bye"
set Label1 . Text to "done"
initialize global var to "hello"
Button1.Click

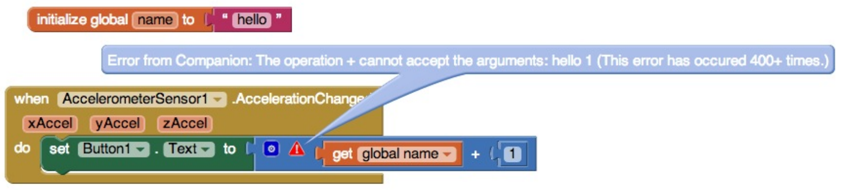
Figure 5-7: If the block that is generating an error is collapsed, the entire group of blocks is expanded.

5.3.2 Quick-fire events

A user might have an error generated by a quick-fire event, such as an accelerometer sensor or a timer with a short interval. (Quick-fire events defined in Ch. 4.) This code will execute multiple times per second, perhaps even thousands, generating the same error repeatedly unless it is stopped. The current version of App Inventor 2 addresses this problem by not sending any information back to the Blocks Editor for five seconds after a block generates an error. Even though the block might still be executing, the error message is only sent once every 5 seconds. This is so the Blocks Editor doesn't get backed up processing error messages. However, 5 seconds is not a very long time for the user to fix the error and she likely does not need the error to appear again so soon. After she dismisses the error, there is not much time before another dialogue box appears, making it hard to change the code behind the error message.

With my implementation of sending errors to blocks, the message does not block the user from continuing to program. When errors are generated, the device keeps track of how many times that block has generated that specific error and only sends it back to the Blocks Editor at certain points. It sends it every time for the first 10 times the error is generated, then the 50th, and then again for every hundred times it happens. It appends the error message to the user with a note about how many times the error has occurred (Fig. 5-8). If it were to send the error back every time it was generated, the Blocks Editor would get backed up dealing with all the *retvals* containing the error messages. However, it is also important to convey to the user that the error is being generated many times. The error message on the block is appended with a message to reflect how many times the error has been generated.

Before my changes, the message that appears on the device was also throttled so that it wouldn't appear constantly if an error was continually firing. The message would stay visible for five seconds and wouldn't appear again for ten. The purpose of the message on the device is to alert the user that something has gone wrong and that she should divert her attention to the blocks editor. I did not change the behavior of the message on the device because the throttling that existed before gave sufficient information to the user while being non-intrusive since the user's attention would be directed to the Blocks Editor.



```
Error from Companion: The operation + cannot accept the arguments: hello 1 (This error has occured 400+ times.)
```

initialize global name to " hello "
when AccelerometerSensor1 .AccelerationChange
xAccel yAccel zAccel
do set Button1 . Text to
get global name + 1

Figure 5-8: An error that is firing rapidly

5.3.3 More Helpful Error Messages

I also changed the error messages that are displayed to the user. I made several improvements to the existing error messages (Fig. 5.1):

- Lists are now represented with square brackets and elements are separated by commas, as they are in Python. Since Python is a popular language for novice programmers to learn, it is helpful for elements of the language to look similar to things they have seen before.
- Strings are now represented with quotation marks, as they are in most text-based languages.
- A clause about type was added to the end of the message to indicate to the user what was wrong about the arguments.

These improvements help in situations where the nature of the data makes the error message hard to understand. For example, the last row of Fig. 5.1 shows a list of lists represented as strings with comma separated values. This is confusing because it just looks like a single list of integers. The improved representation makes it clear that it is a list of strings.

Before	**After**
Strings represented as: `Hello`	`"Hello"`
Lists represented as: `(0 2 3 4)`	`[0, 2, 3, 4]`
`The operation + cannot accept the arguments: 0 hello`	`The operation + cannot accept the arguments: [0, "hello"]. '+' requires two numbers as input and at least one of yours is the wrong type.`
`Select list item: Attempt to get list item number 3 of a list of length 2: (foo bar)`	`Select list item: Attempt to get list item number 3 of a list of length 2: ["foo", "bar"]`
`(1,2,3,4 2,3,4,5 3,4,5,6 4,5,6,7)`	`["1,2,3,4", "2,3,4,5", "3,4,5,6", "4,5,6,7"]`

Table 5.1: Improvements made to error messages

5.3.4 Evaluation

My implementation of errors on blocks slows down processes running on the device in certain situations. Quick-fire events cause the app to become unresponsive to the phone. Because of the augmented code, the time it takes for each piece of code to execute is slightly longer. While a quick-fire event runs, there is evidence that the app is still executing the code, but the behavior on the screen does not reflect what the app is doing. The bit of code in Fig. 5-9 would be expected to cause a label to increment at a constant rate. However, the display doesn't update often enough to show every value, but rather increments in random intervals. The device will eventually display a message that the running app has become unresponsive. This shows that the app is able to keep up and execute the code, but it doesn't have time to communicate changes to the device's display. On every device, there is a watchdog app that makes sure everything is running smoothly. The watchdog app sends requests to the running App Inventor app and expects responses. The app is not actually unresponsive, but doesn't have time to respond to the watchdog app on the phone that it is still running because it is too busy evaluating the augmented code. Possible solutions are lengthening the amount of time between the firing of quick-fire events or changing the sensitivity settings of the device so the event doesn't fire as often. It also might just have to be a compromise that sacrifices performance for desired features.

Another unresolved issue with the current implementation occurs when a user has a quick-fire event that toggles between firing two or more different errors. Because it is never the same error multiple times in a row, the device will think it is a new error each time and send it to the Blocks Editor each time the event fires. This is problematic for two reasons: (1) if the errors being sent back to the Blocks Editor are not throttled, the Blocks Editor won't have time to process them all

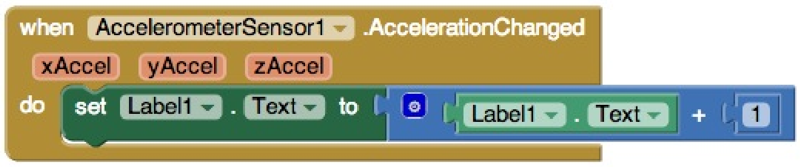


Figure 5-9: A quick-fire event

and will get backed up and (2) the user is unaware of how many times the error has been generated. Solving this problem requires keeping track of more information about which blocks generate errors on the device.

The goal with the error counts was to give the user as clear an idea as possible of what is happening without directly processing every detail. The count on the block conveys that the errors are still generating without directly processing each error. This model doesn't work with the throttling, however, because it does not accurately represent what is going on. It fails to represent the fact that the toggled errors are being generated multiple times.

Chapter 6

Watch

Logic errors occur when a user's program behaves differently from how she expects it to. Possible scenarios include an event not happening when she expects or something displaying an unexpected value. It is hard to implement features to detect logic errors because it is hard to know what the intention of the user is. However, it is possible to provide tools to help the user find the source of her logic errors. A standard tool in any programming language is a feature that allows users to print information to a console during execution. In App Inventor, while there is no built in console for the user to print to, a user can create her own feedback mechanism by including a label in the interface of her app that she can use to display any information from the code she wishes to view. However, this requires using an extra component to display the information and extra blocks to specify the information to be displayed. It is also not likely that a novice programmer would think to do this. My solution is an extension of the *watch* feature in App Inventor Classic that allows users to see the values of their variables during execution.

6.1 My Version of Watch

My implementation of the *watch* feature in App Inventor 2 is similar to *watch* in App Inventor Classic (Sec. 2.3.2), but it shows the user all the values of the expression, as opposed to just the last value (Fig. 6-1). Now, if the user is debugging her program, she can see how the expression changed. This is helpful in debugging scenarios in which an unexpected value occurs. If the watched block is in a loop, it would also give an indication of how many times the loop is iterating, which might be helpful information. An educator might also use a *watch* to explain to a student how the program is executing. The feature is much more helpful than using a label to show this information because the user can get the information faster. It also makes more sense than using a label because the information is displayed near the code, rather than as part of the interface of the app she is writing.

The values from the expression are displayed with the most recent values at the bottom. A survey of programmers and non-programmers revealed that this is the favored way to represent the output in the available space (Sec. 6.2). My work done so far is the simplest implementation of a better designed *watch*, discussed further in Sec. 8.2.

My implementation of *watch* (Appendix B) is very similar to my implementation of augmented code for putting errors on blocks, discussed in Sec. 5.3. If a user watches a block, the block is wrapped with a `watch` tag along with the block ID (Fig. 6-2, Fig. 6-3). When the interpreter on the device executes the watched code, it sends the result of the expression back to the Blocks Editor. When the Blocks Editor receives the result of a watched expression, it appends the result to the list of returned values and displays it in the comment square on the block in order.

The implementation of *watch* is similar to the implementation of augmented code, but the two are orthogonal. They do not need the other to be implemented to work, but they could work at the same time.

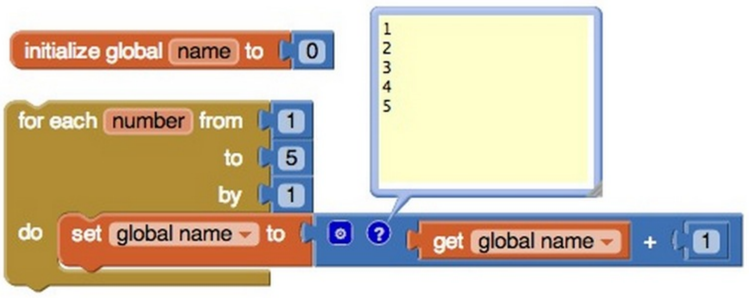


Figure 6-1: The result of watching the addition block during a loop that increments a variable

```
(forrange $number
    (begin (set-var! g$count
           (call-yail-primitive + (*list-for-runtime* (get-var g$count) 1))
           '(number number) "+"))
    1 5 1)
```

Figure 6-2: Code from Fig. 6-1 without `watch` tag

```
(forrange $number
    (begin (set-var! g$count
           (watch 87 (call-yail-primitive + (*list-for-runtime* (get-var g$count) 1))
           '(number number) "+")))
    1 5 1)
```

Figure 6-3: Code from Fig. 6-1 with *watch* tag

6.2 User Study

When displaying information to users, it is important to consider the way the information is presented. There are a number of different ways the result of a watched expression could be shown to the user. The values could be displayed horizontally or vertically and in different orders. The values could be separated by commas, lines, or just by space.

Fig. 6-4 shows a vertical display of values with the most recent value at the top. The advantage of this is that when the user looks at the watch bubble, she reads the most recent value first and then proceed to read the others. If all the values occur during one execution, this makes sense for the user. The disadvantage is that it doesn't match what happens when output from print statements is printed to a console. The expectation among programmers is that time goes down in a sequence of printed commands.

Fig. 6-5 shows a vertical display of values with the most recent value at the bottom. This is intuitive because it behaves similarly to a console that contains the output of print statements. The most recent values appear below the previous values. The disadvantage of this is that users have to scroll down to see the recent values rather than just having them on top.

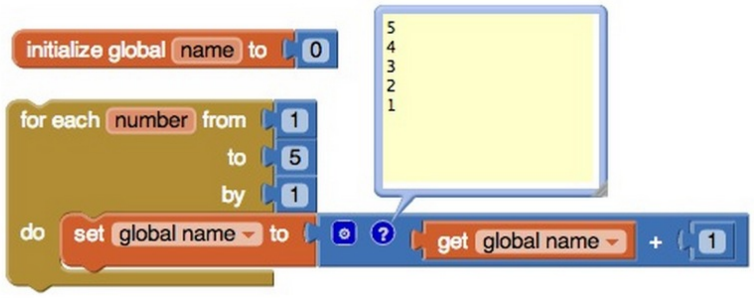
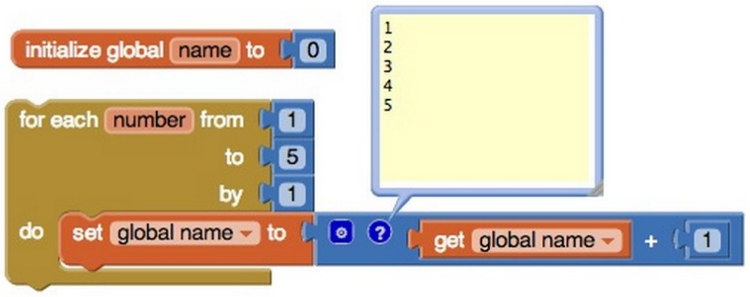


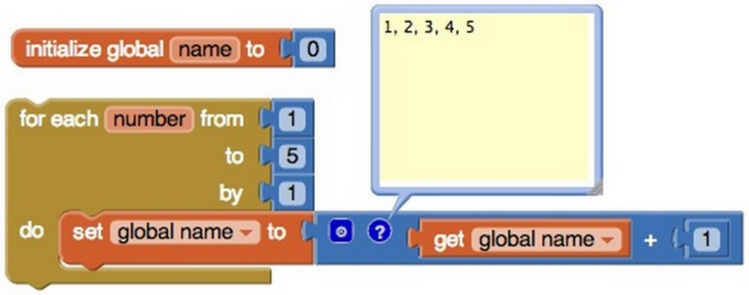
Figure 6-4: Displayed vertically with most recent at the top

Fig. 6-6 and Fig. 6-7 show horizontal lists. Horizontal display raises the same question as vertical display of what order the values should be in. There is also the question of how to separate the values (Fig. 6-8). The values might be strings that contain commas or semicolons themselves, in which



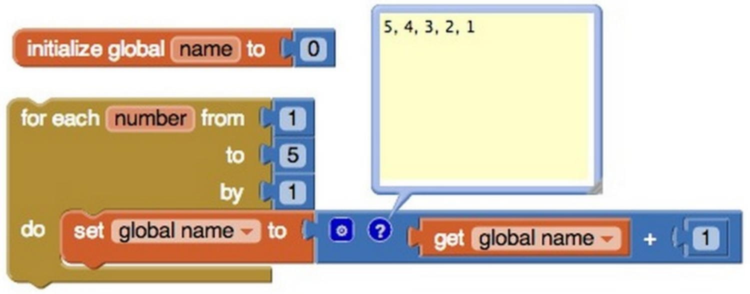
```
1
2
3
4
5
```

Figure 6-5: Displayed vertically with most recent at the bottom



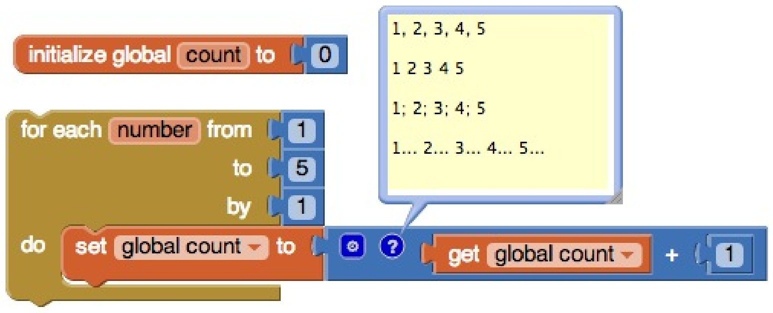
```
1, 2, 3, 4, 5
```

Figure 6-6: Displayed horizontally with most recent on the right



```
5, 4, 3, 2, 1
```

Figure 6-7: Displayed horizontally with most recent on the left


initialize global count to 0
for each number from 1 to 5 by 1
do set global count to
get global count + 1

```
1, 2, 3, 4, 5

1 2 3 4 5

1; 2; 3; 4; 5

1... 2... 3... 4... 5...
```

Figure 6-8: Alternate ways to represent values horizontally

Model	No programming experience	Programming experience
Vertical, most recent at top	3	1
Vertical, most recent at bottom	4	4
Horizontal, most recent on right	2	0
Horizontal, most recent on left	1	0
Other	0	0

Table 6.1: Results of User Study

case this would be confusing. Lists are also often displayed with commas separating the values, so if an expression was returning a list, the elements in the list might not be clear.

An informal survey was done to gain insight into how users feel about these different representations. A total of 15 students were surveyed: five who had some programming background, and 10 who had not (Fig. 6.1). They were given the above figures (Fig. 6-4, Fig. 6-5, Fig. 6-6, Fig. 6-7, Fig. 6-8) as well as a blank box for if they had other ideas. They were asked which was the most intuitive or if they had a better way they would like the data represented. Both groups, with and without a programming background, lean towards vertical with the most recent value at the bottom. Several people said that there must be a better way, but they weren't sure what it would look like.

Chapter 7

Error Reports

When a user is in live development mode, error messages are recorded using ACRA error reporting. Error reports are automatically sent to a cloud database, via an HTTP POST or PUT request from the phone. Such a report includes the error message, the time of the error, and the device on which it was generated, identified by a unique ID. I have access to the ACRA database and have done preliminary analysis of the types and frequencies of error messages in App Inventor, discussed in Sec. 4.

The error reports contain a field for a unique ID, which is generated upon installation of the app and functions as a user ID within the error database. However, is not linked to the user ID in the cloud where the projects are stored. Thus, the stored error reports currently cannot be linked back to the App Inventor user who generated them or the project from which they were generated.

7.1 Attempted Solution: Saving Code

My first solution was to have the phone send the YAIL code for the current screen to the error database with the error report. The goal was to add the augmented code of the program (discussed in Sec. 5.3) to the error report along with the block ID of the block causing the error. Then, when analyzing the error reports, we would know what sort of program the error came from. We would also be able to track what the user does if she generates the same error multiple times. She may change the code, or she might just be running the code that is generating the same error repeatedly.

Since the app is always running the most recent YAIL code, it makes sense for the device to send the augmented code when it sends the error report. However, when the YAIL code is sent from the Blocks Editor to the device, the device does not save a readable version of the code. When the device receives YAIL code from the Blocks Editor, it distributes it among various environments and event registries. The code is stored as objects, rather than human-readable code. Thus, if I want to send the most recent version of the code with the error report, I have to save it somewhere else on the device.

I implemented the means to save all code that gets sent to the device in a human-readable format. Every time blocks in the Blocks Editor are added or changed, all the blocks are compiled to YAIL. For every top level block, if any blocks within that block were added or changed, that top level block is sent to the device. My implementation includes storing a list of tuples on the device, where the first element is the block ID of a top level block and the second element is a string version of the YAIL code that corresponds to that block. When the device receives any code, it creates the block ID and code tuple and appends it to the list. The list of tuples then not only shows the current state of the code, but also shows what the user did to get to that state. This entire list of tuples is then sent with the error report. This allows us to analyze changes users make and gives valuable insight into how they understand the language. Since we see everything they tried to do, it shows us where they may need additional support. The problem with this implementation is that the list of tuples gets long very quickly and there is only a limited amount of space on the device. Kawa also does not have a quick way to convert a list to a string, so when the error report is sent with the list of tuples, it takes too long to convert the tuple list to a format that can be sent.

Because storing all the changed code was taking up too much space on the device, I decided to store only the most recent version of the code. My second design still used the list of tuples. In this implementation, every time new code comes to the device, it walks down the list of tuples and replaces the tuple that corresponds to the changed block and replaces its code. This makes the list much shorter than if we were to store every change the user makes. However, if the user has a large project, the list will still get very long and will have to be limited at some point. Simply counting the number of tuples in the list is not enough because the user might have lots of blocks under one top level block. Another problem with this design is that searching through the list for the correct tuple takes a long time. Environments containing the top level block IDs and the associated YAIL code could have been implemented, but instead I opted for a better design.

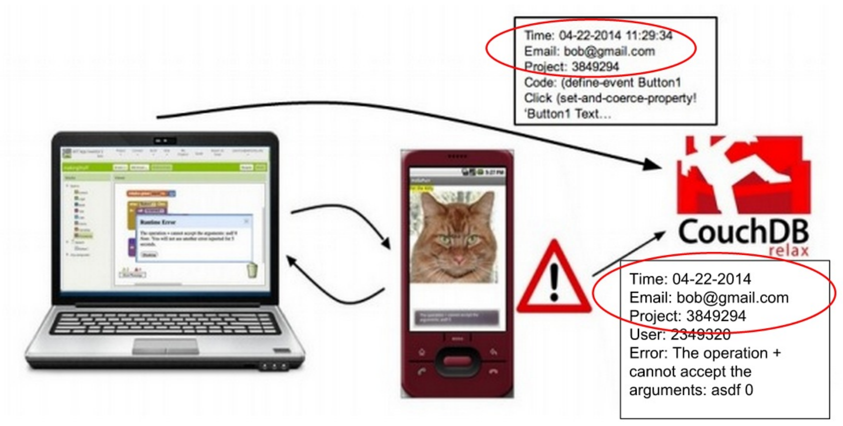
7.2 Solution: Stored User Information

My improved and implemented design (Appendix C) is to send the user email and project ID from the device with the error report (Fig. 7-1). Email addresses and project IDs are unique, so these two pieces of information are enough to link an error with the project. The implementation of this involved sending the user email and project ID to the device when the app is first loaded. The part of App Inventor that runs in the browser is implemented with Java and JavaScript. The Blocks Editor, the blocks themselves, and the communications between the Blocks Editor and device are written in JavaScript. The Java side takes care of the project storage for App Inventor. The user email and project ID are stored on the Java side, but the Blocks Editor needs to access them with JavaScript to send them to the device. To allow these variables to be accessed from JavaScript involved defining a function that exports the desired information from Java to JavaScript.

Having the user email and project ID is helpful because we can go to where App Inventor projects are stored and find out everything about that user. We could test hypotheses about which type of App Inventor users get errors: beginner, intermediate, or advanced. We have survey data from many App Inventor 2 users that includes information about their background and there might be a correlation. I also previously did work to understand how people use App Inventor blocks [OT13]. For each project and each user, I counted the number of blocks, number of components, number of projects, how many programming constructs such as procedures and global variables they use, and how they use them. I then used clustering to break users into groups based on programming experience and App Inventor proficiency. Matching a user who was not included in the clustering to a cluster will give an idea of their level as a programmer and we can test hypotheses about how programming level corresponds to runtime errors.

However, App Inventor only stores the current state of the program. If we go to the App Inventor database to look for the project that generated the error, it is likely to have changed since the error was generated. To solve this, there is another component to the design that has not been implemented yet. The proposed solution is to send a snapshot of the block code in an error report from the Blocks Editor along with the user email, project ID, and time stamp. For every error, there will be two entries in the error database that can be linked by a common user email, project ID, and time stamp. This will once again allow us to see what the user changes in between times the error is generated, but will not take up storage or execution time on the device since the block code is sent from the Blocks Editor.

An important consideration when sending data is the privacy of that data. The user emails and project IDs are sensitive because they can be used to uniquely identify a user. The projects themselves also might contain sensitive information if, for example, the user wrote a navigation app that uses her home address. This has not been implemented yet, but all data being sent to ACRA should be encrypted to prevent information leaks.


Time: 04-22-2014 11:29:34
Email: bob@gmail.com
Project: 3849294
Code: (define-event Button1
Click (set-and-coerce-property!
'Button1 Text…

Time: 04-22-2014
Email: bob@gmail.com
Project: 3849294
User: 2349320
Error: The operation +
cannot accept the
arguments: asdf 0

Figure 7-1: When an error is generated, reports are sent from both the Blocks Editor and the device.

Chapter 8

Conclusion

8.1 Summary of work

App Inventor programs generate runtime and behavioral errors, but the environment does not provide much support for the users to debug them. My previous work lead to the conclusion that many users get stuck on one particular error for a while before finally fixing it or giving up. My work has been to implement tools to help the users find the source of runtime errors and understand the state of variables and expressions over time, as well as implement the means for App Inventor developers to better understand what users do when they encounter errors.

Previously, when a user encountered a runtime error, a dialogue box would appear on the screen in front of the code, which the user would have to dismiss before debugging the error. The flaw with this model was that the user couldn't see the error and the code at the same time. My solution is to display the error on the block that generated the error. This not only allows the user to see the error and debug at the same time, but also gives her more insight into the source of the error. I also improved the error messages themselves, changing the way data is represented and giving more information about why the error occurred. Implementing the mechanism that puts errors on blocks required understanding the communication between the Blocks Editor and the connected device and augmenting the YAIL code sent from the Blocks Editor to the device with `augment` tags and block IDs. (Ch. A)

My help for users' behavioral errors centered around trying to help them understand the state of their program. App Inventor Classic had a feature called *watch* that allowed a user to view the most recent value of an expression after the execution of the program. App Inventor 2 did not have such a feature, so I implemented and extended what App Inventor Classic had. In my version, not only is the most recent value displayed, but all intermediate values of that expression. This is more helpful than just the most recent value because it allows the user to see when something unexpected may have happened. Implementing *watch* involved adding a `watch` tag to the YAIL code for blocks the user is watching when the YAIL code is sent from the Blocks Editor to the device. (Ch. B)

I also implemented the means to collect more detailed information when errors are generated. Previously, reports containing the error message, information about the device it was generated on, and the time were sent to a database, but there was no way to link the errors with the users or projects who generated them. I implemented the means to send the user and project ID with the error reports, allowing future work to be done to make corrolations between types of users and errors. (Ch. 7)

8.2 Future work

There are a few ways my work needs to be finished. It would be nice to clean *watch* up so that users are only allowed to watch blocks that it makes sense to watch because some blocks do not return values. For example, a statement such as setting the text of a label is executed without returning anything. If a novice user is allowed to put a *watch* on this block, she may be confused about why no value is returned. One solution would be to not allow users to watch statement blocks. Another option is to allow statement blocks to be watched, but have them return something that indicates they were executed, such as the word ”done”. I also need to implement a new icon for watched blocks. Currently, the icon is a comment block and the result of the *watch* is displayed in a comment block. This prevents the user from putting an actual comment on the block and from looking at the Blocks Editor, it is not clear whether a block has a comment or is being watched.

Better user studies are necessary to understand to measure the how my debugging features help users. Studies with beginner, intermediate, and advanced users would give insight to who needs the most support and whether they need different types of help. And of course, performance costs need to be assessed and determined if the desired features are worth the compromise. A project could be to measure the exact extent to which augmented code slows down execution.

My project lays the groundwork for future development of smarter and more complex debugging support. A feature that would be very helpful for finding behavioral errors would be having the block that is being executed outlined while the app is running. There could also be a single stepping mode like Scratch and StarLogo TNG (Sec. 2.2) where users can execute one block at a time. The implementation of augment would make these features possible because the code executing on the device is now linked to the source block during runtime.

The current implementation of *watch* is only the first step towards a more helpful way to keep track of and display the values of variables and expressions. Rather than displaying a static list of the history of the variable, the user could be given control over time. As the user slides over the history of the variable, the code that changed the variable to that value could be outlined (Fig. 8-1).

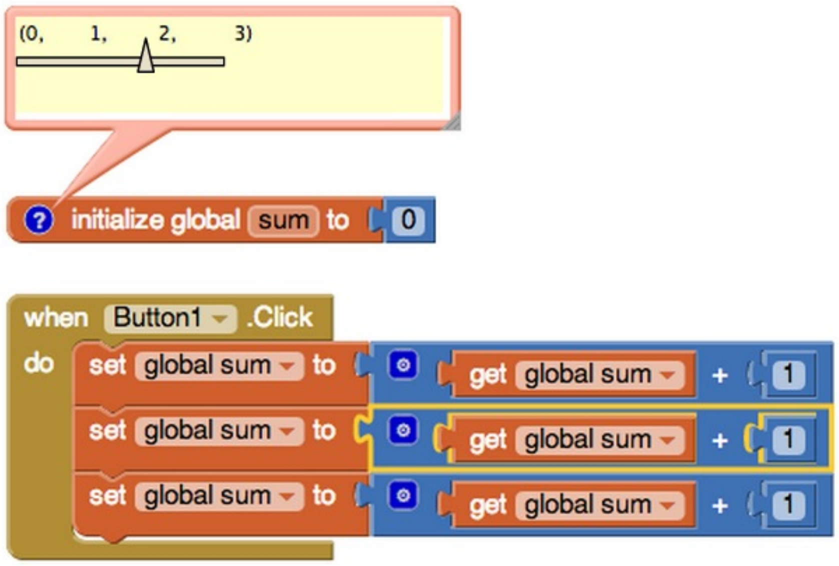

(0, 1, 2, 3)
initialize global sum to 0
when Button1 .Click
do set global sum to get global sum + 1
set global sum to get global sum + 1
set global sum to get global sum + 1

Figure 8-1: The designer window

It would also be useful to keep track of all variables without the user having to ask or specify what she wants recorded, like Scratch (Sec. 2.2.1). This could be extended so that when a runtime error occurs, any information about the execution of the program and the state of variables and values of expression blocks that might be useful to the user would already be available. Otherwise, the user has to specify which variables she wants to watch and rerun the program, making it crash in the same place. A future project could be to implement the means to store all information while the program executes and display it to the user effectively (Sec. 2.1.2). Rather than a time slider for an individual variable, it could be global and show how all the variables change and what blocks are being executed. As the user manipulates the time slider, the blocks that were being executed at that time would be outlined, like a Deterministic Replay Debugger (Sec. 2.1.2).

There is also room for work to understand how users construct their programs. We understand the current state of projects, but haven't looked into how users create them. This would involve setting up the means to store every action the user performs in the Blocks Editor and Designer.

We'd be interested in seeing the common things people do wrong. Many people use App Inventor as an introduction to computer programming, and every language has common mistakes that are especially easy for novices to make. Knowing what these mistakes are in App Inventor would help educators know what they need to explain and also might give insight into ways the App Inventor language could be clarified or where some provided guidance is necessary for new users. Benjamin Xie has submitted a proposal to implement the means of collecting more fine grained data of program construction and using this data to understand user errors and the use of blocks [Xie14]. There is also a proposal for an extended version of this that records all actions an App Inventor user takes and provides an interface for them to replay the execution of the program [TM14]. Users will also be able to share the steps they have taken for someone else to look at and provide help.

More data could also be collected when error reports are sent to ACRA. We know the error and we can track how the code changes in between iterations of that error, but we don't know if the user ever fixes the error. If the error doesn't show up in the reports anymore, it means that either the user fixed it, or she gave up and just isn't running the program anymore. The later is much more concerning than the former and would indicate the need for more debugging support. We don't want users getting frustrated with their errors to the point of giving up on App Inventor.

In both App Inventor Classic and App Inventor 2, if a user plugs a block of the wrong type into a socket, the block immediately jumps out. In App Inventor Classic, a message is also displayed, but not in App Inventor 2. Some sort of support should be provided to users in App Inventor 2 who get code-time errors.

Fine grained tracking of App Inventor program construction would also provide data that could be used to implement an intelligent tutor for App Inventor. When users encounter code time or runtime errors, the next steps they take could be recorded. We have a notion of the users' proficiency level from the clustering done on data from all the programs. We could run machine learning algorithms on the steps done after encountering these problems, favoring more advanced users' solutions, and learn how to solve common issues. Then when someone encounters the same issue and can't seem to figure it out, the intelligent tutor could give suggestions based on what other people did. This would be helpful for code-time errors. If a user repeatedly tries to do something wrong, support would be provided.

Bibliography

[AC11] S. Andrica and G. Candea. "WaRR: A tool for high-fidelity web application record and replay". In: *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. 2011, pp. 403–410. DOI: `10.1109/DSN.2011.5958253`.

[Ai1] MIT Center for Mobile Learning, App Inventor Classic home page, `http://explore.appinventor.mit.edu/classic`, accessed Feb. 24, 2014.

[Ai2] MIT Center for Mobile Learning, App Inventor 2 home page, `http://appinventor.mit.edu`, accessed Feb. 24, 2014.

[BDM97] Ron Baecker, Chris DiGiano, and Aaron Marcus. "Software Visualization for Debugging". In: *Commun. ACM* 40.4 (Apr. 1997), pp. 44–54.

[Gom+13] Lorenzo Gomez et al. "RERAN: Timing- and Touch-sensitive Record and Replay for Android". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 72–81. ISBN: 978-1-4673-3076-3. URL: `http://dl.acm.org/citation.cfm?id=2486788.2486799`.

[Lie97] Henry Lieberman. "The Debugging Scandal and What to Do About It". In: *Commun. ACM* 40.4 (Apr. 1997), pp. 27–29.

[MEH10] James Mickens, Jeremy Elson, and Jon Howell. "Mugshot: Deterministic Capture and Replay for JavaScript Applications". In: *Symp. on Networked Systems Design and Implementation*. 2010.

[OT13] Johanna Okerlund and Franklyn Turbak. *A Preliminary Analysis of App Inventor Blocks Programs*. Poster presented at Visual Languages and Human Centric Computing (VLHCC), Sept. 15-19, San Jose, California, http://cs.wellesley.edu/ tinkerblocks/VLHCC13-abstract.pdf, http://cs.wellesley.edu/ tinkerblocks/VLHCC13-poster.pdf. 2013.

[Roq06] Ricarose Roque. "A Debugger for Starlogo TNG: Detecting and Removing Bugs in a Visual Programming Language". Thesis proposal. 2006.

[Scr] Scratch project, MIT Lifelong Kindergarten Group, `http://scratch.mit.edu/`, accessed Feb 24, 2014.

[Sen+13] Koushik Sen et al. "Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: ACM, 2013, pp. 488–498. ISBN: 978-1-4503-2237-9. DOI: `10.1145/2491411.2491447`. URL: `http://doi.acm.org/10.1145/2491411.2491447`.

[Sta] StarLogo TNG project, MIT Scheller Teacher Education Program, `http://education.mit.edu/projects/starlogo-tng`, accessed Feb 24, 2014.

[TM14] Franklyn Turbak and Eni Mustafaraj. *Codelines: Replaying, Sharing, and Annotating App Inventor Project Histories*. April, 2014 Google Research Awards proposal. 2014.

[ULF97] David Ungar, Henry Lieberman, and Christopher Fry. "Debugging and the Experience of Immediacy". In: *Commun. ACM* 40.4 (Apr. 1997), pp. 38–43.

[Xie14] Benjamin Xie. *Request for SuperUROP Support of Blocks-Level Analytics with App Inventor*. Proposal for SuperUROP at MIT. 2014.

Appendix A

Errors on Blocks

A.1 Blocks converted to YAIL

```
/**
 * appinventor/lib/blockly/src/core/generator.js
 * Generate code for the specified block (and attached blocks).
 * @param {Blockly.Block} block The block to generate code for.
 * @return {string|!Array} For statement blocks, the generated code.
 *     For value blocks, an array containing the generated code and an
 *     operator order value.  Returns '' if block is null.
 * Johanna: added augment tag and block ID
 */
Blockly.CodeGenerator.prototype.blockToCode = function(block) {
  if (!block) {
    return '';
  }
  if (block.disabled) {
    // Skip past this block if it is disabled.
    var nextBlock = block.nextConnection && block.nextConnection.targetBlock();
    return this.blockToCode(nextBlock);
  }

  var func = this[block.type];
  if (!func) {
    throw 'Language "' + this.name_ + '" does not know how to generate code ' +
        'for block type "' + block.type + '".';
  }
  var code = func.call(block);
  if (code instanceof Array) {
    // Value blocks return tuples of code and operator order.
    //[12.2.13] added augment tag to keep track of block id when yail is sent to device
    var result = [this.scrub_(block, "(augment " + block.id + " " + code[0] + ")"), code[1]];
    return result;
  } else {
    var result = this.scrub_(block, "(augment " + block.id + " " + code + ")");
    return result;
  }
};
```

A.2 Define syntax for augment

```
;;appinventor/buildserver/src/com/google/appinventor/buildserver/resources/runtime.scm
;;(On the device)
;;Augment: keep track of block ID of currently executed block
(define-syntax augment
    (syntax-rules()
      ((_ info exp)
          (let ((ans (with-current-block-id info (lambda () exp))))
              (after-execution info)
              ans))))
```

A.3 Helper functions for define syntax for augment

```
;;appinventor/buildserver/src/com/google/appinventor/buildserver/resources/runtime.scm
;;(On the device)
;;Check to see if block had previously generated error
;;If so, remove error.
(define (after-execution block-id)
  (if (member block-id (*:get-blocks-with-errors *this-form*))
    (begin (send-to-block block-id (list "OK" "noError")) ;;Alert Blocks Editor that the error
           (*:remove-from-last-block-with-error *this-form* block-id) ;;No longer has to count
           (*:remove-from-errors *this-form* block-id))) ;;Block is no longer flagged for havin
  )

;;appinventor/buildserver/src/com/google/appinventor/buildserver/resources/runtime.scm
;;(On the device)
;;Keep track of block id of currently executed code
(define (with-current-block-id block-id thunk)
    (let ((old-block-id (*:get-current-block-id *this-form*)))
      (*:set-current-block-id *this-form* block-id)
      (let ((result (thunk)))
        (*:set-current-block-id *this-form* old-block-id) ;; Reset block-id to remembered value
        result)))
```

A.4 Send error to Blocks Editor

```
//appinventor/components/src/com/google/appinventor/components/runtime/util/RetValManager.java
//On Blocks editor side:
//Add argument to sendError function for app on device to call
//Creates retval for type error
public static void sendError(String error, String blockid) {
  synchronized (semaphore) {
    JSONObject retval = new JSONObject();
    try {
      retval.put("status", "OK");
      retval.put("type", "error");
      retval.put("value", error);
      retval.put("blockid", blockid);
    } catch (JSONException e) {
      Log.e(LOG_TAG, "Error building retval", e);
      return;
    }
    boolean sendNotify = currentArray.isEmpty();
    currentArray.add(retval);
    if (sendNotify) {
      semaphore.notifyAll();
    }
  }
}
```

```
;;appinventor/buildserver/src/com/google/appinventor/buildserver/resources/runtime.scm
;;Send error and block ID to Blocks Editor when error occurs
;;Calls sendError(String error, String blockid)
(define (send-error error)
  (add-to-blocks-with-errors current-block-id)
  (let ((ans (add-to-blocks-plus-errors current-block-id error blocks-plus-errors)))
    (if (or (equal? ans "newError") (equal? ans "newBlock")) ;;only send error to blocks editor
      (com.google.appinventor.components.runtime.util.RetValManager:sendError error current-blo
      (let ((count (cadr ans)))
        (if (or (equal? count 50) (equal? (modulo count 100) 0)) ;; send if equal to 50 and mu
          (com.google.appinventor.components.runtime.util.RetValManager:sendError
              (string-append error " (This error has occured " (number->string count) "+ times
```

A.5 Blocks Editor process errors

```
//appinventor/blocklyeditor/src/replmgr.js
Blockly.ReplMgr.processRetvals = function(responses) {
    var block;
    for (var i = 0; i < responses.length; i++) {
        var r = responses[i];
        console.log("processRetVals: " + JSON.stringify(r));
        switch(r.type) {
        case "return":
            if (r.blockid != "-1") {
                block = Blockly.mainWorkspace.getBlockById(r.blockid);
                if (r.status == "OK") {
                    block.replError = null;
                    if (r.value && ((r.value != '*nothing*') && (r.value != "noError"))) { //A
                    this.setDoitResult(block, r.value);
                    }
                } else {
                    if (r.value) {
                        block.replError = "Error from Companion: " + r.value;
                    } else {
                        block.replError = "Error from Companion";
                    }
                }
            }
```

```
                break;
            case "pushScreen":
                var success = window.parent.BlocklyPanel_pushScreen(r.screen);
                if (!success) {
                    console.log("processRetVals: Invalid Screen: " + r.screen);
                }
                break;
            case "popScreen":
                window.parent.BlocklyPanel_popScreen();
                break;
            case "error":
                console.log("Blockly.ReplMgr.processRetvals: error for block " + r.blockid); //JOHA
                if (r.blockid != -1) {
                    block = Blockly.mainWorkspace.getBlockById(r.blockid);
                    block.replError = "Error from Companion: " + r.value;

                    block.setErrorIconText("Error from Companion " + r.value);
                    var rootBlock;
                    var current = block;
                    do {
                        // uncollapse collapsed blocks
                        if (current.collapsed){
                            current.setCollapsed(false);
                        }
                        rootBlock = current;
                        current = rootBlock.parentBlock_;
                    } while (current);

                    block.errorIcon.setVisible(true);

                } else { // Not associated with particular block; put up Jeff's dialog
                    if (!this.runtimeError) {
                        this.runtimeError = new goog.ui.Dialog(null, true);
                    }
                    if (this.runtimeError.isVisible()) {
                        this.runtimeError.setVisible(false);
                    }
                    this.runtimeError.setTitle("Runtime Error");
                    this.runtimeError.setButtonSet(new goog.ui.Dialog.ButtonSet().
                                                   addButton({caption:"Dismiss"}, false, true));
                    this.runtimeError.setContent(r.value + "<br/><i>Note:</i>&nbsp;You will not see
                    this.runtimeError.setVisible(true);
                }
            }
        }
        Blockly.WarningHandler.checkAllBlocksForWarningsAndErrors();
    };
```

Appendix B

Watch

B.1 Added watch tag during YAIL code generation

```
/**
 * /Users/johannaokerlund/app-inventor-gerrit/appinventor/blocklyeditor/src/replmgr.js:
 * Generate code for the specified block (and attached blocks).
 * @param {Blockly.Block} block The block to generate code for.
 * @return {string|!Array} For statement blocks, the generated code.
 *     For value blocks, an array containing the generated code and an
 *     operator order value.  Returns '' if block is null.
 * Johanna: added watch tag (in addition to augment tag)
 */
Blockly.CodeGenerator.prototype.blockToCode = function(block) {
  if (!block) {
    return '';
  }
  if (block.disabled) {
    // Skip past this block if it is disabled.
    var nextBlock = block.nextConnection && block.nextConnection.targetBlock();
    return this.blockToCode(nextBlock);
  }

  var func = this[block.type];
  if (!func) {
    throw 'Language "' + this.name_ + '" does not know how to generate code ' +
        'for block type "' + block.type + '".';
  }
  var code = func.call(block);
  if (code instanceof Array) {
    if (block.watch){
      var result = [this.scrub_(block, "(augment " + block.id + " (watch " + block.id + " " + c
      code[1]];
      return result;
    } else {
    // Value blocks return tuples of code and operator order.
    var result = [this.scrub_(block, "(augment " + block.id + " " + code[0] + ")"), code[1]];
    return result;
  }
  }else {
    result = this.scrub_(block, "(augment " + block.id + " " + code + ")");
    return result;
  }
};
```

B.2 Define syntax for watch

```
;;appinventor/buildserver/src/com/google/appinventor/buildserver/resources/runtime.scm
;;Execute the block and send result back to the Blocks Editor
(define-syntax watch
  (syntax-rules()
    ((_ info exp)
      (let ((result exp))
        (send-to-block info (list "OK" result)) ;;Send result to Blocks Editor
        result)))) ;;Return the result
```

B.3 Process

```
\\appinventor/blocklyeditor/src/replmgr.js
\\Added case to check if block is being watched
\\If it is, then place returned value on the block
Blockly.ReplMgr.processRetvals = function(responses) {
    var block;
    for (var i = 0; i < responses.length; i++) {
        var r = responses[i];
        console.log("processRetVals: " + JSON.stringify(r));
        switch(r.type) {
        case "return":
            if (r.blockid != "-1") {
                block = Blockly.mainWorkspace.getBlockById(r.blockid);
                if (r.status == "OK") {
                    block.replError = null;
                    if (r.value && (r.value != '*nothing*')) {
                        if (block.watch){
                            this.appendToWatchResult(block,r.value);
                        } else {
                        this.setDoitResult(block, r.value);
                        }
                    }
                } else {
                    if (r.value) {
                        block.replError = "Error from Companion: " + r.value;
                    } else {
                        block.replError = "Error from Companion";
                    }
                }
            }
            break;
        case "pushScreen":
            var success = window.parent.BlocklyPanel_pushScreen(r.screen);
            if (!success) {
                console.log("processRetVals: Invalid Screen: " + r.screen);
            }
            break;
        case "popScreen":
            window.parent.BlocklyPanel_popScreen();
            break;
        case "error":
            if (!this.runtimeError) {
                this.runtimeError = new goog.ui.Dialog(null, true);
            }
            if (this.runtimeError.isVisible()) {
                this.runtimeError.setVisible(false);
            }
            this.runtimeError.setTitle("Runtime Error");
            this.runtimeError.setButtonSet(new goog.ui.Dialog.ButtonSet().
                                           addButton({caption:"Dismiss"}, false, true));
            this.runtimeError.setContent(r.value + "<br/><i>Note:</i>&nbsp;You will not see an
            this.runtimeError.setVisible(true);
        }
    }
    Blockly.WarningHandler.checkAllBlocksForWarningsAndErrors();
};


Blockly.ReplMgr.appendToWatchResult = function(block, value) {
    var comment = "";
    if (block.comment) {
        comment = block.comment.getText();
    }
    comment = value + "\n" + comment;
    if (block.comment) {
        block.comment.setVisible(false);
    }
    block.setCommentText(comment);
    block.comment.setVisible(true);
}
```

Appendix C

Extra Information to ACRA

C.1 Set up means to get User email and Project ID to JavaScript

```
// appinventor/appengine/src/com/google/appinventor/client/editor/youngandroid/BlocklyPanel.j
// Methods to export code from Java to JavaScript.
// Added code to return user email, ID, and project ID
public static String getCurrentUserEmail(){
  return Ode.getInstance().getUser().getUserEmail();
}

public static String getCurrentUserId(){
  return Ode.getInstance().getUser().getUserId();
}

public static String getCurrentProjectId(){
  return "" + Ode.getInstance().getCurrentYoungAndroidProjectId();
}

private static native void exportMethodsToJavascript() /*-{
  $wnd.BlocklyPanel_sendError =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::sendError(Ljava/
  $wnd.BlocklyPanel_getCurrentUserEmail =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::getCurrentUserEm
  $wnd.BlocklyPanel_getCurrentUserId =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::getCurrentUserId
  $wnd.BlocklyPanel_getCurrentProjectId =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::getCurrentProjec
  $wnd.BlocklyPanel_initBlocksArea =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::initBlocksArea(L
  $wnd.BlocklyPanel_blocklyWorkspaceChanged =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::blocklyWorkspaceC
  $wnd.BlocklyPanel_checkWarningState =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::checkWarningState
  $wnd.BlocklyPanel_callToggleWarning =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::callToggleWarning
  $wnd.BlocklyPanel_checkIsAdmin =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::checkIsAdmin());
  $wnd.BlocklyPanel_indicateDisconnect =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::indicateDisconnec
  // Note: above lines are longer than 100 chars but I'm not sure whether they can be split
  $wnd.BlocklyPanel_pushScreen =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::pushScreen(Ljava/
  $wnd.BlocklyPanel_popScreen =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::popScreen());
  $wnd.BlocklyPanel_createDialog =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::createDialog(Ljav
  $wnd.BlocklyPanel_hideDialog =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::HideDialog(Lcom/g
  $wnd.BlocklyPanel_setDialogContent =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::SetDialogContent(
  $wnd.BlocklyPanel_getComponentInstanceTypeName =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::getComponentInsta
  $wnd.BlocklyPanel_getComponentInfo =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::getComponentInfo(
  $wnd.BlocklyPanel_getComponentsJSONString =
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::getComponentsJSO
  $wnd.BlocklyPanel_getYaVersion=
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::getYaVersion());
  $wnd.BlocklyPanel_getBlocksLanguageVersion=
    $entry(@com.google.appinventor.client.editor.youngandroid.BlocklyPanel::getBlocksLanguage

}-*/;
```

C.2 Build user info into YAIL and send to device

```
// appinventor/blocklyeditor/src/replmgr.js
Blockly.ReplMgr.buildYail = function() {
  console.log("BUILD YAIL"); //JOHANNA
  var phoneState;
  var code = [];
  var blocks;
  var block;
  var needinitialize = false;
  if (!window.parent.ReplState.phoneState) { // If there is no phone state, make some!
      window.parent.ReplState.phoneState = {};
  }
  phoneState = window.parent.ReplState.phoneState;
  if (!phoneState.formJson || !phoneState.packageName)
      return;                 // Nothing we can do without these
  if (!phoneState.initialized) {
      phoneState.initialized = true;
      phoneState.blockYail = {};
      phoneState.componentYail = "";
  }

  var jsonObject = JSON.parse(phoneState.formJson);
  var formProperties;
  var formName;
  if (jsonObject.Properties) {
      formProperties = jsonObject.Properties;
      formName = formProperties.$Name;
```

```
    }
    var componentMap = Blockly.Component.buildComponentMap([], [], false, false);
    var componentNames = [];
    for (var comp in componentMap.components)
        componentNames.push(comp);
    if (formProperties) {
        if (formName != 'Screen1')
            code.push(Blockly.Yail.getComponentRenameString("Screen1", formName));
        var sourceType = jsonObject.Source;
        if (sourceType == "Form") {
            code = code.concat(Blockly.Yail.getComponentLines(formName, formProperties, null /
        } else {
            throw "Source type " + sourceType + " is invalid.";
        }

        // Fetch all of the components in the form, this may result in duplicates
        componentNames = Blockly.Yail.getDeepNames(formProperties, componentNames);
        // Remove the duplicates
        var uniqueNames = componentNames.filter(function(elem, pos) {
            return componentNames.indexOf(elem) == pos;});
        componentNames = uniqueNames;

        code = code.join('\n');

        if (phoneState.componentYail != code) {
            // We need to send all of the comonent cruft (sorry)
            needinitialize = true;
            phoneState.blockYail = {}; // Sorry, have to send the blocks again.
            this.putYail(Blockly.Yail.YAIL_CLEAR_FORM);
            this.putYail(code);
            this.putYail("(user-email " + window.parent.BlocklyPanel_getCurrentUserEmail() + "
            this.putYail("(user-id " + window.parent.BlocklyPanel_getCurrentUserID() + ")");
            this.putYail("(project-id " + window.parent.BlocklyPanel_getCurrentProjectID() + "
            this.putYail(Blockly.Yail.YAIL_INIT_RUNTIME);
            console.log(code);
            console.log(window.parent.BlocklyPanel_getCurrentUserEmail());
            phoneState.componentYail = code;
        }
    }
```

C.3 Device stores information

```
;;appinventor/buildserver/src/com/google/appinventor/buildserver/resources/runtime.scm
(define-syntax user-email
  (syntax-rules()
    ((_ email)
        (*:set-userEmail *this-form* email)
        )))

(define-syntax user-id
  (syntax-rules()
    ((_ email)
        (*:set-userEmail *this-form* email)
        )))

(define-syntax project-id
  (syntax-rules()
    ((_ email)
        (*:set-userEmail *this-form* email)
        )))
```

C.4 Error Occurs and Device sends information with error report

```
;;appinventor/buildserver/src/com/google/appinventor/buildserver/resources/runtime.scm
(define (process-exception ex)
       (define-alias YailRuntimeError <com.google.appinventor.components.runtime.errors.YailR
       ;; The call below is a no-op unless we are in the wireless repl
       (com.google.appinventor.components.runtime.ReplApplication:reportError ex get-userEma
       (if isrepl
           (when ((this):toastAllowed)
                 (begin (send-error (ex:getMessage))
                        ((android.widget.Toast:makeText (this) (ex:getMessage) 5):show)))

           (com.google.appinventor.components.runtime.util.RuntimeErrorAlert:alert
            (this)
            (ex:getMessage)
            (if (instance? ex YailRuntimeError) ((as YailRuntimeError ex):getErrorType) "Run
            "End Application")))
```

C.5 ACRA sends extra information in error report

```
// appinventor/components/src/com/google/appinventor/components/runtime/ReplApplication.java
public static void reportError(Throwable ex, String email, String userId, String projectId) 
  if (thisInstance != null && thisInstance.active)
    ACRA.getErrorReporter().putCustomData("userEmail", email);
    ACRA.getErrorReporter().putCustomData("userId", userId);
    ACRA.getErrorReporter().putCustomData("projectId", projectId);
    ACRA.getErrorReporter().handleException(ex);
}
```