# AI-Augmented Interface for Incremental App Development in MIT App Inventor

by

Ashley Granquist

B.S. Computer Science and Engineering, MIT, 2024

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

© 2024 Ashley Granquist. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Ashley Granquist
Department of Electrical Engineering and Computer Science
May 10, 2024

Certified by: Harold Abelson
Class of 1922 Professor of Computer Science and Engineering, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

# AI-Augmented Interface for Incremental App Development in MIT App Inventor

by

Ashley Granquist

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

## ABSTRACT

The recent revolutionary advancements in Artificial Intelligence (AI) have presented immense opportunities and challenges in computer science education. This thesis presents the development of an AI-powered tool built on top of MIT App Inventor to help students incrementally design mobile applications. The tool allows students to describe desired changes to their MIT App Inventor mobile applications in natural language and have those changes be implemented automatically. Students can alternate between manually editing their app and using this tool, enabling them to collaborate with AI and incrementally develop apps with a degree of assistance from AI that meets their needs and is appropriate for their skill level and workflow preferences. This thesis also explores the benefits and detriments of such a tool, as well as observations and lessons learned from studying the ways students interact with the tool during a pilot study.

Thesis supervisor: Harold Abelson
Title: Class of 1922 Professor of Computer Science and Engineering

# Acknowledgments

I would like to thank my thesis supervisor, Hal Abelson, for his guidance and support as well as the opportunity to work in his lab. I would like to thank Evan Patton, MIT App Inventor's Lead Software Developer, for his assistance with the technical aspects of the project both in terms of setting up Aptly's overall foundations and answering questions. I would like to thank David Kim, Software Developer, for his help with designing and executing the study with students, as well as his encouragement throughout the project. I would also like to thank the many students who worked on Aptly, and in particular, Arianna Scott, who contributed dozens of examples of editing apps which were used to help GPT-4 create the Aptly code representing the modified app for each change.

# Contents

**Title page** 1

**Abstract** 3

**Acknowledgments** 5

**List of Figures** 9

**List of Tables** 11

**1 Introduction** 13

1.1 MIT App Inventor . . . . . . . . . . . . . . . . . . . . . . . . . . 14
1.2 AI-powered programming assistants . . . . . . . . . . . . . . . . . . . . 17
1.2.1 GitHub Copilot . . . . . . . . . . . . . . . . . . . . . . . . 18
1.2.2 ChatGPT, Gemini, and other LLMs . . . . . . . . . . . . . . . . . . 18
1.2.3 Aptly for generating MIT App Inventor apps . . . . . . . . . . . . . . . 18
1.3 Aptly Editing . . . . . . . . . . . . . . . . . . . . . . . . . . 20

**2 Aptly Editing Scenario Walk-Through** 23

2.1 Example usage . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.2 High-level technical approach . . . . . . . . . . . . . . . . . . . . . 37

**3 Methodology** 39

3.1 Background regarding Aptly . . . . . . . . . . . . . . . . . . . . . . 39
3.1.1 Aptly code . . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.1.2 Abstract Syntax Tree (AST) representation . . . . . . . . . . . . . . 44
3.2 Aptly Editing Implementation Details . . . . . . . . . . . . . . . . . . 45
3.2.1 User workflow for Aptly Editing . . . . . . . . . . . . . . . . . . . 45
3.2.2 Editing Technical Process Overview . . . . . . . . . . . . . . . . . . 46
3.2.3 The Pencil and Undo buttons . . . . . . . . . . . . . . . . . . . . 49
3.2.4 Aptly Project Editor Wizard . . . . . . . . . . . . . . . . . . . . 50
3.2.5 Generating Aptly code representing the current app . . . . . . . . . . . 50
3.2.6 Ranking Examples . . . . . . . . . . . . . . . . . . . . . . . . 50

3.2.7 Processing GPT-4's output . . . 52
3.2.8 Computing edit operations using Zhang-Shasha (ZSS) algorithm . . . 52
3.2.9 Creating edit events . . . 60
3.2.10 Applying edit events to the project . . . 88

**4 Study Design 91**
4.1 Participants . . . 91
4.2 Survey Questions . . . 92
4.3 Learning the basics of the tool . . . 92
4.4 Task 1: Replicate a specific app . . . 93
4.5 Task 2: Create an app to solve a given problem . . . 93
4.6 Task 3: Create an app of the participants' choosing . . . 94

**5 Results & Discussion 95**
5.1 Summary of participants' initial attitudes towards programming . . . 95
5.2 Task 1 (Recreating an app) . . . 97
5.3 Task 2 (Solving a given problem) . . . 100
5.4 Task 3 (Creating any app) . . . 103
5.5 General observations . . . 104
5.6 Success with incremental development . . . 106
5.7 Changes in participants' attitudes towards programming and AI . . . 108
5.8 Conclusion . . . 109

**References 111**

# List of Figures

1.1 The Designer tab in MIT App Inventor, where users can assemble the user interface of their app. Currently, the 'TodoListLabel' component is selected and the user is viewing the editable properties for this component. . . . . . . 15
1.2 The Blocks tab in MIT App Inventor, where users can assemble code blocks to program the functionality of their app. Currently, the 'AddButton' component is selected and the user is viewing available blocks associated with that component. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
1.3 The code blocks for an app to add three user-provided inputs and display the sum when a button is clicked. . . . . . . . . . . . . . . . . . . . . . . . . . 17
1.4 The Aptly App Creation interface, where users provide a natural language description of their desired app. The user has provided a name for their app as well as a description, and clicked the "Code it!" button. The text-based code on the right will be used to construct an MIT App Inventor app once the user clicks the "Make it!" button. . . . . . . . . . . . . . . . . . . . . . . 19
1.5 The Designer tab after the user has provided the TodoList app description and clicked "Make it!". . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
1.6 The Blocks tab after the user has provided the TodoList app description and clicked "Make it!". . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

2.1 An app description for 'TrashTracker,' an app to help a student track trash in her neighborhood. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.2 The app description and Aptly code for "TrashTracker" after the user has clicked "Code It!" . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.3 The mock screen for "TrashTracker" after the user has clicked "Make It!" . . 25
2.4 The blocks editor for "TrashTracker" after the user has clicked "Make It!" . . 26
2.5 The edit description entered to help the student keep track of plastic bags. . 27
2.6 The updated mock screen with an additional button to help the student keep track of plastic bags. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.7 The updated blocks editor with code for an additional button to help the student keep track of plastic bags. . . . . . . . . . . . . . . . . . . . . . . . 29
2.8 The student instructs Aptly Editing "Add instructions at the top of the screen so other people will know how to use the app." . . . . . . . . . . . . . . . . . 30

2.9 The result from the previous command. . . . . . . . . . . . . . . . . . . . . . 31
2.10 The student manually changing the label text. . . . . . . . . . . . . . . . . . . 32
2.11 The command "Make everything bigger" is currently being processed by Aptly Editing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
2.12 The result from the previous command, with the top label being cut off on the right. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.13 The student in the process of undo-ing Aptly Editing's change. . . . . . . . . 35
2.14 Aptly's edit has successfully been undone. . . . . . . . . . . . . . . . . . . . 36

3.1 The mock screen for an app that allows users to select a shape, color, and size to draw in. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.2 The code blocks for an app that allows users to select a shape, color, and size to draw in. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.3 The Aptly Project Editor Wizard. In this example, the user has clicked the blue icon with the pencil icon and typed a description of their desired edit into the input text box. The undo button is disabled because the user has not yet made a change using Aptly. . . . . . . . . . . . . . . . . . . . . . . . 47
3.4 The editing technical process . . . . . . . . . . . . . . . . . . . . . . . . . . 48
3.5 The button text change reflected at the AST level. Matched nodes are shown in white and Updated nodes are shown in purple. . . . . . . . . . . . . . . . 55
3.6 The bound component event change reflected at the AST level. Matched nodes are shown in white and Updated nodes are shown in purple. . . . . . . 59
3.7 A code sample from an app involving a list. . . . . . . . . . . . . . . . . . . 71
3.8 Adding an element to the middle of the list, shown at the AST level. . . . . 72
3.9 The same code sample from the previous figure with 'Ashley' inserted into the middle of the list. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
3.10 A code sample from an app involving a conditional statement. . . . . . . . . 74
3.11 The AST representation of a bound component event containing a conditional statement. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
3.12 The updated code sample from an app involving a conditional statement. . . 76
3.13 The AST node changes for the bound component event involving a conditional statement. Insertions are shown in green, removals are shown in red, and updates are shown in purple. . . . . . . . . . . . . . . . . . . . . . . . . . . 78
3.14 Code blocks for an app that uses a procedure to update a button's text . . . 81
3.15 The AST representations of the original and modified app, with the insertions in green. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
3.16 The AST representations of the original and modified app, with the deletions in red and the insertion in green. . . . . . . . . . . . . . . . . . . . . . . . . 86
3.17 The Aptly Project Edit Undo Wizard . . . . . . . . . . . . . . . . . . . . . . 89

# List of Tables

5.1 Pre-Study Survey: Responses to "How interested (1-5) would you be in learning how to program?" . . . . . 95
5.2 Pre-Study Survey: Responses to "Given the right tools, could you see yourself making your own interesting app by yourself?" . . . . . 95
5.3 Pre-Study Survey: Responses to "What would you say is the biggest factor that stops most people from learning programming?" . . . . . 96
5.4 Pre-Study Survey: Sentiment of responses to "How strongly would you agree with this statement: 'Programming is accessible for everyone.' " . . . . . 96

# Chapter 1

# Introduction

The movement to democratize programming and improve coding's accessibility has existed for over a decade [1] [2]. With the recent revolutionary advancements in Artificial Intelligence (AI), the development of AI-powered tools has accelerated this movement and provided new opportunities to further expand programming's accessibility to individuals without a formal computer science education. This thesis presents Aptly Editing, an AI-powered tool built on top of MIT App Inventor to help students incrementally design mobile applications. The tool allows students to describe desired changes to their MIT App Inventor mobile applications in natural language and have those changes be implemented automatically. Students can alternate between manually editing their app and using this tool, enabling them to collaborate with AI and incrementally develop apps with a degree of assistance from AI that meets their needs and is appropriate for their skill level and workflow preferences.

The rest of this introduction will introduce MIT App Inventor, a drag-and-drop blocks-based programming platform aimed at democratizing programming, as well as a few other AI-powered programming assistants. After providing this context, the end of this introduction will present Aptly Editing and its specific contributions to improving the accessibility of programming.

## 1.1 MIT App Inventor

MIT App Inventor is a blocks-based, drag-and-drop programming platform for developing mobile applications. MIT App Inventor was developed with the goal of democratizing programming and enables users to create functional, highly-customized apps without needing to study the complexities and syntactic details of any particular text-based programming language.

MIT App Inventor's project editor has two tabs, the "Designer" and "Blocks" tabs, for editing the user interface and functionality of the app, respectively. In the "Designer" tab, users can drag components such as buttons, labels, and images onto the screen to design the user interface of their app, and they can also select non-visible components such as sounds and sensors. The palette on the left lists components available for use in the app, grouped by category, such as "User Interface" for general visible components, "Media" for components such as sound players, text-to-speech, and sound recorders, and "Storage" for files, spreadsheets, and database components. Users can click on a given component they've added to their user interface in order to update its properties (such as text, font size, width, visibility, and color). Figure 1.1 shows what the Designer tab looks like for an example 'TodoList' app.

In the "Blocks" tab, users can assemble code blocks to construct the functionality of their app, such as choosing to change the background color of the app when a button is clicked. The user can click on the name of a given component to select blocks related to that specific component (such as event handlers for when a button is clicked, and setters and getters for the button's properties), and there are also generic blocks such as conditional statements, comparators, lists, dictionaries, variables, and blocks for defining and calling procedures (i.e. functions). Figure 1.2 shows what the Blocks tab looks like for the same sample To-Do List app.

MIT App Inventor allows students to practice programming without needing to worry

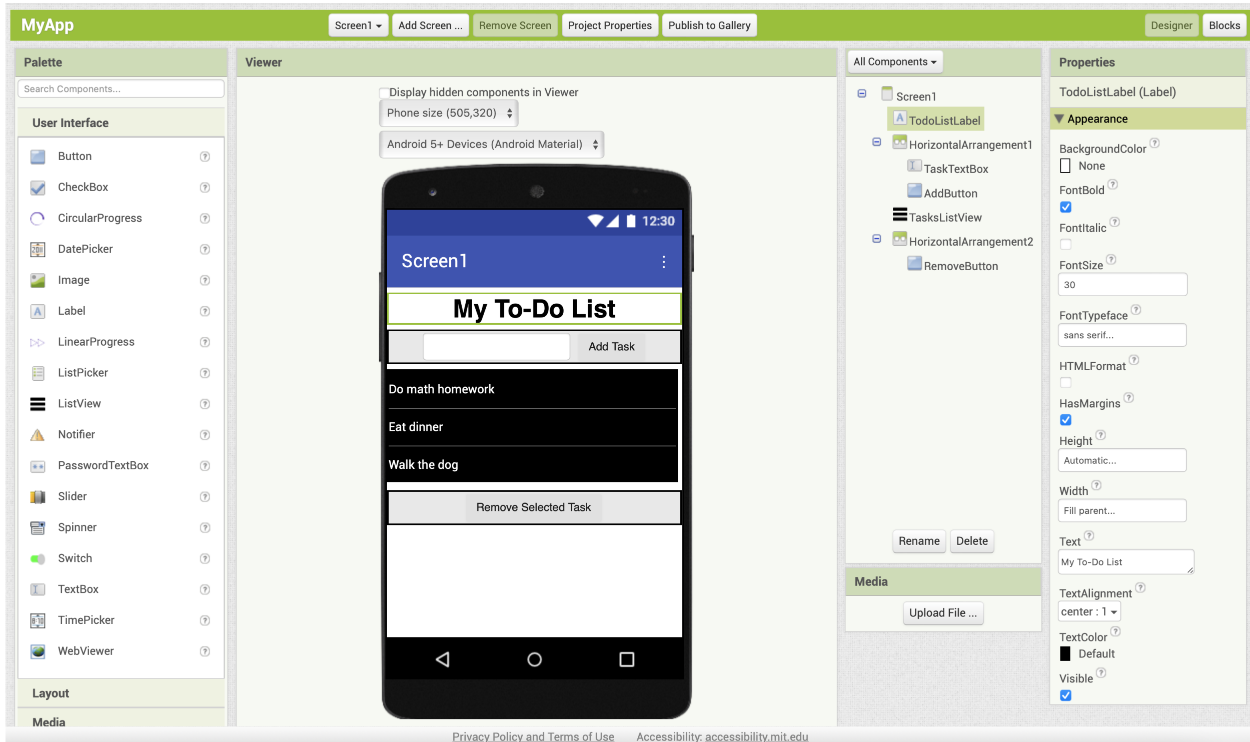


Figure 1.1: The Designer tab in MIT App Inventor, where users can assemble the user interface of their app. Currently, the 'TodoListLabel' component is selected and the user is viewing the editable properties for this component.

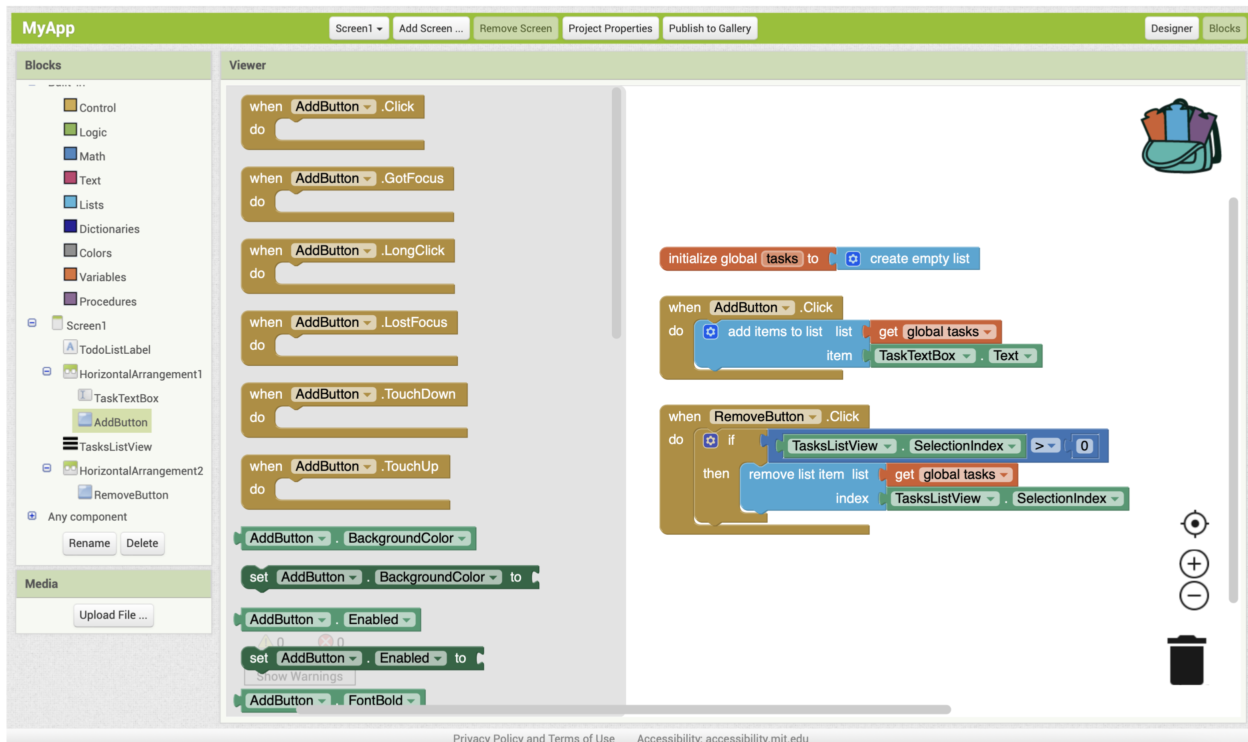


Figure 1.2: The Blocks tab in MIT App Inventor, where users can assemble code blocks to program the functionality of their app. Currently, the 'AddButton' component is selected and the user is viewing available blocks associated with that component.

about the syntactic details of a text-based programming language. Students can learn about important programming concepts such as fields, variables, conditional statements, and functions/procedures/methods, as well as data structures such as lists and dictionaries, in a beginner-friendly environment.

For example, in figure 1.3, the user has defined a procedure called `addThreeNumbers` which returns the sum of three inputs. The user is utilizing this procedure in order to calculate the sum of the numbers inputted into three text boxes. The block where the procedure is called automatically has the parameter names included, so the user can easily remember which arguments they need to include. (It is unlikely that a programmer would choose to use a procedure in this case when they could just inline the sum calculation sum; this particular procedure is merely used as an example of how a student might learn about writing and calling functions with arguments).

Figure 1.3: The code blocks for an app to add three user-provided inputs and display the sum when a button is clicked.

## 1.2 AI-powered programming assistants

In recent years, revolutionary advancements in AI have led to the development and proliferation of AI-powered programming assistants, from conversational AI agents to code completion tools. Several examples are introduced here.

### 1.2.1 GitHub Copilot

GitHub Copilot [3] is a tool for text-based programming languages such as Python, JavaScript, and C++ that can provide suggested auto-completions of code that a programmer is actively writing, as well as provide suggested implementations of code to match descriptions a programmer has provided.

Copilot is widely used by programmers to code more efficiently; utilizing Copilot means programmers no longer need to turn to a web search to double check syntax, and can simply accept suggestions rather than continuing to type out every character.

However, Copilot may be less suitable for beginners, who would need to become familiar with the specific syntax of a text-based programming language in order to effectively evaluate Copilot's suggestions.

### 1.2.2 ChatGPT, Gemini, and other LLMs

Since its release in 2022, ChatGPT [4] has been widely used for programming assistance. Common use cases are generating code as a starting point for projects, providing ChatGPT with buggy code and requesting debugging assistance, and asking ChatGPT for insight regarding general programming concepts or syntax questions.

Other chatbots based on Large Language Models (LLMs) such as Google's Gemini [5] are utilized in similar contexts.

A limitation of these LLM-based chatbots is similar to that of Copilot; since these tools focus on text-based code, beginners may have a difficult time evaluating an AI assistant's code suggestions.

### 1.2.3 Aptly for generating MIT App Inventor apps

Several members of the MIT App Inventor team, including myself, have been working on developing functionality to allow users to create MIT App Inventor apps by providing a

natural language description of their desired app [6] [7] [8].

Users of this tool begin at the interface shown in figure 1.4. In the example shown, the user provides an app name and a natural language description of their desired app, and upon clicking the "Code it!" button and subsequently the "Make it!" button, the user will find that an MIT App Inventor app matching their description has been created for them (figures 1.5 and 1.6).

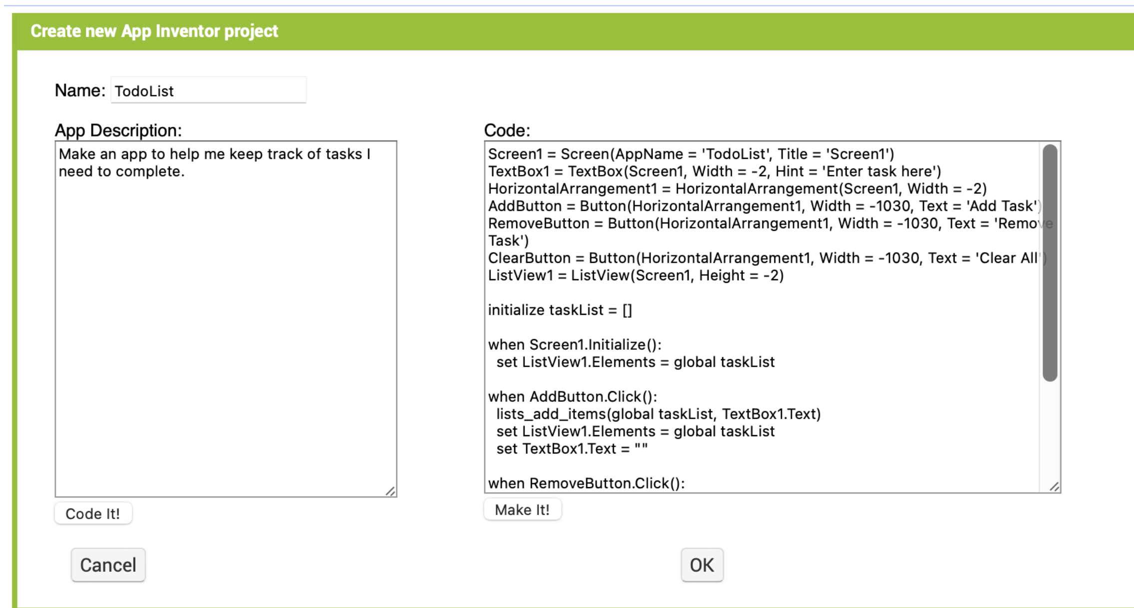


Figure 1.4: The Aptly App Creation interface, where users provide a natural language description of their desired app. The user has provided a name for their app as well as a description, and clicked the "Code it!" button. The text-based code on the right will be used to construct an MIT App Inventor app once the user clicks the "Make it!" button.

The tool provides users with a starting point for their app based on the natural language description they provide, and users are then free to manually enhance the app's user interface and functionality.

The implementation details of this tool (hereafter referred to as Aptly App Creation), as relevant to my work developing Aptly Editing, are discussed in Chapter 3.

Aptly App Creation leaves users with the initially-created app which they are free to edit manually. Without Aptly Editing, discussed in the next section, users would be left to manually complete their app without the further assistance of AI.

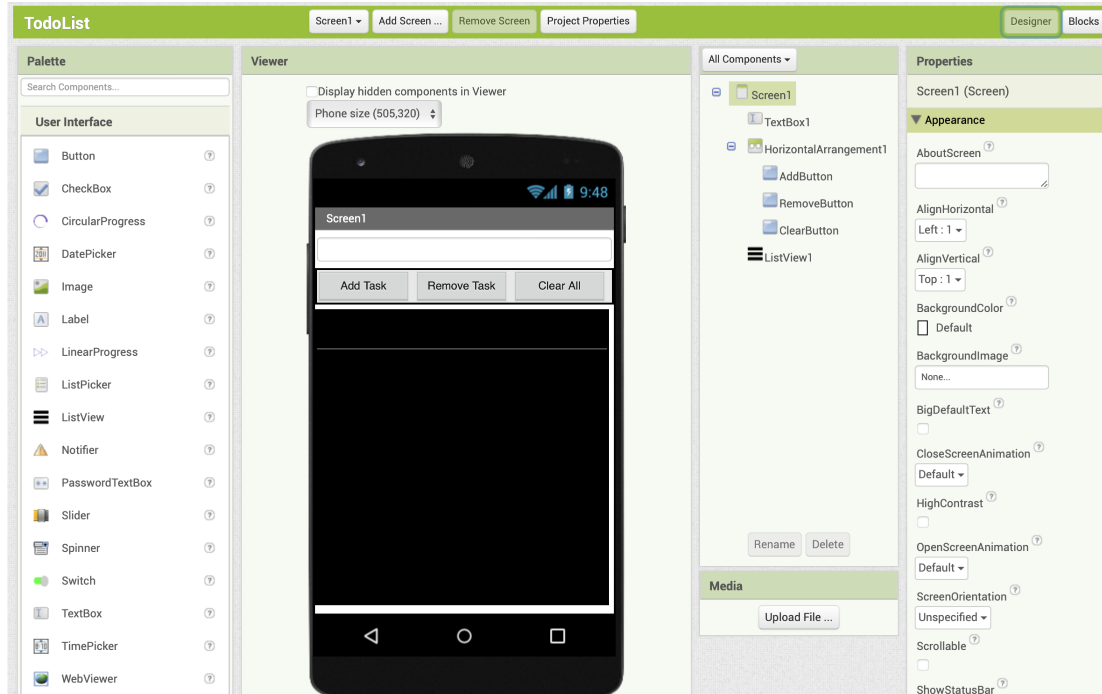


Figure 1.5: The Designer tab after the user has provided the TodoList app description and clicked "Make it!".

## 1.3 Aptly Editing

This thesis presents Aptly Editing, an AI-powered tool built on top of MIT App Inventor to help students incrementally design mobile applications. Users can begin with an MIT App Inventor app that they've created either by themselves or with the help of Aptly App Creation. Then, users can incrementally edit their app by providing natural language descriptions of their desired changes. Users can also choose to edit their app manually, which enables users to effectively collaborate with the AI assistant. In this way, Aptly Editing not only encourages and supports incremental development, but also allows users to utilize the AI assistant to different extents, depending on their workflow preferences and desired level of independence.

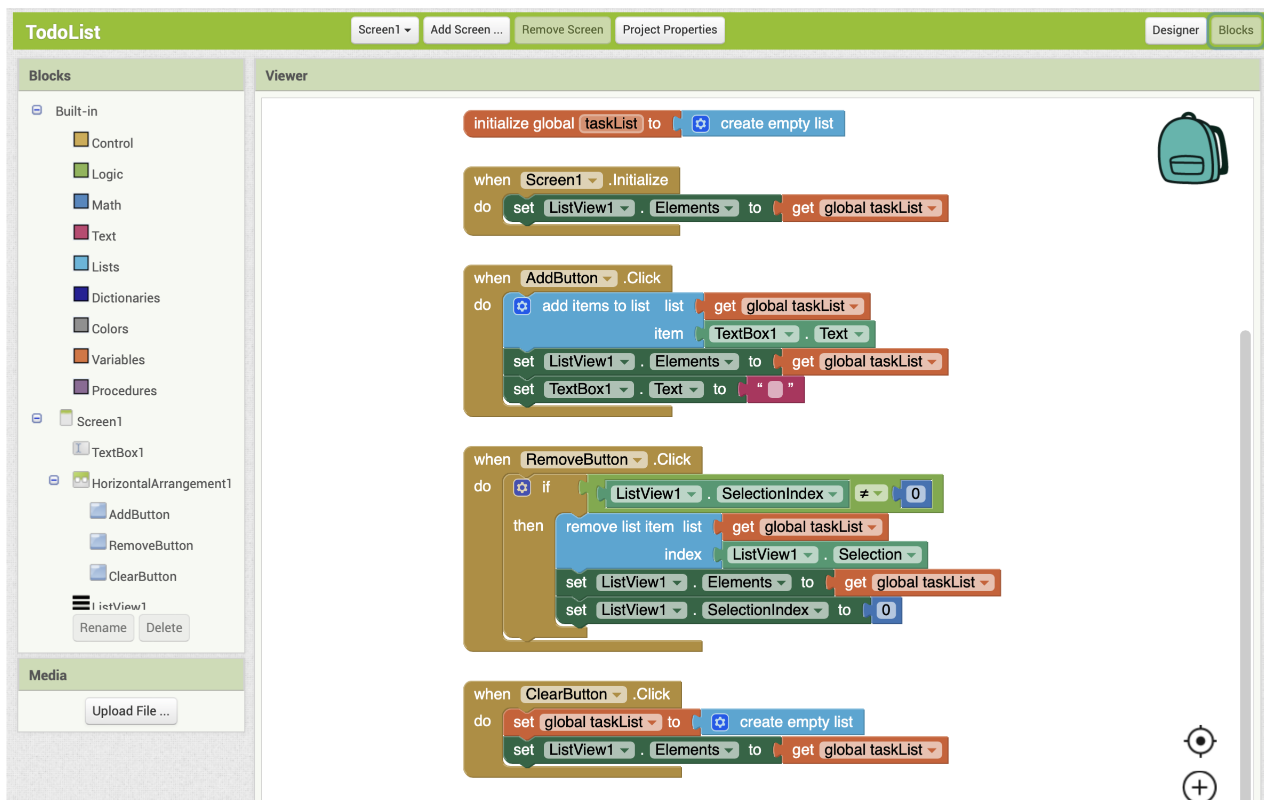


Figure 1.6: The Blocks tab after the user has provided the TodoList app description and clicked “Make it!”.

# Chapter 2

# Aptly Editing Scenario Walk-Through

## 2.1 Example usage

This section presents an example scenario in which a student might want to use Aptly Editing and demonstrates how that student might interact with the tool.

Suppose a student named Abby is concerned about the environment and wants to investigate the magnitude of the littering problem in her neighborhood. She wants to walk through her neighborhood and keep track of how many trash items she sees, but she is worried about losing count if she counts in her head, and she doesn't want to have to carry around a notebook and pen to make tally marks. She decides to make a mobile app to help her keep track of the amount of trash she sees.

Abby begins by using Aptly App Creation, with the following command: "Make an app with buttons for plastic bottles and soda cans. I will click the plastic bottle button when I see a plastic bottle, and I will click the soda can button when I see a soda can. I want the app to keep track of the number of each I've seen." (figure 2.1).

Abby then clicks the "Code It!" button, and sees the text-based representation of an app starting with `Screen1 = Screen...` (figure 2.2) which will be explained in the next section.

She then clicks "Make it!" and sees an MIT App Inventor app matching her description

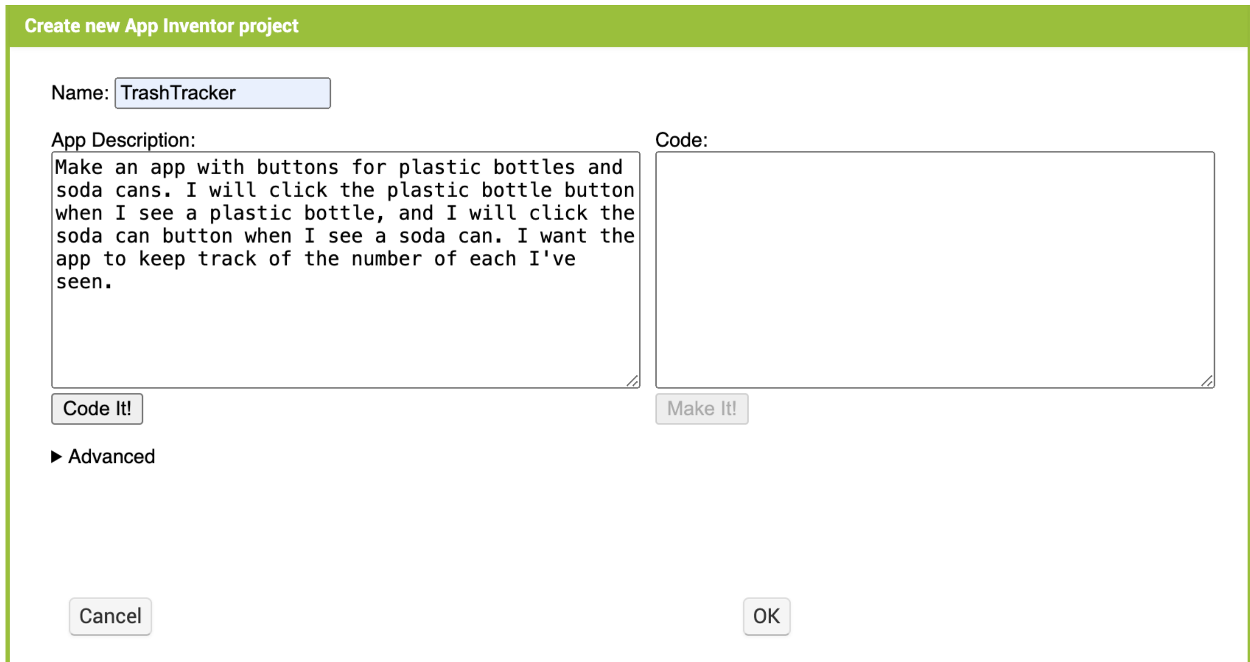


Figure 2.1: An app description for 'TrashTracker,' an app to help a student track trash in her neighborhood.

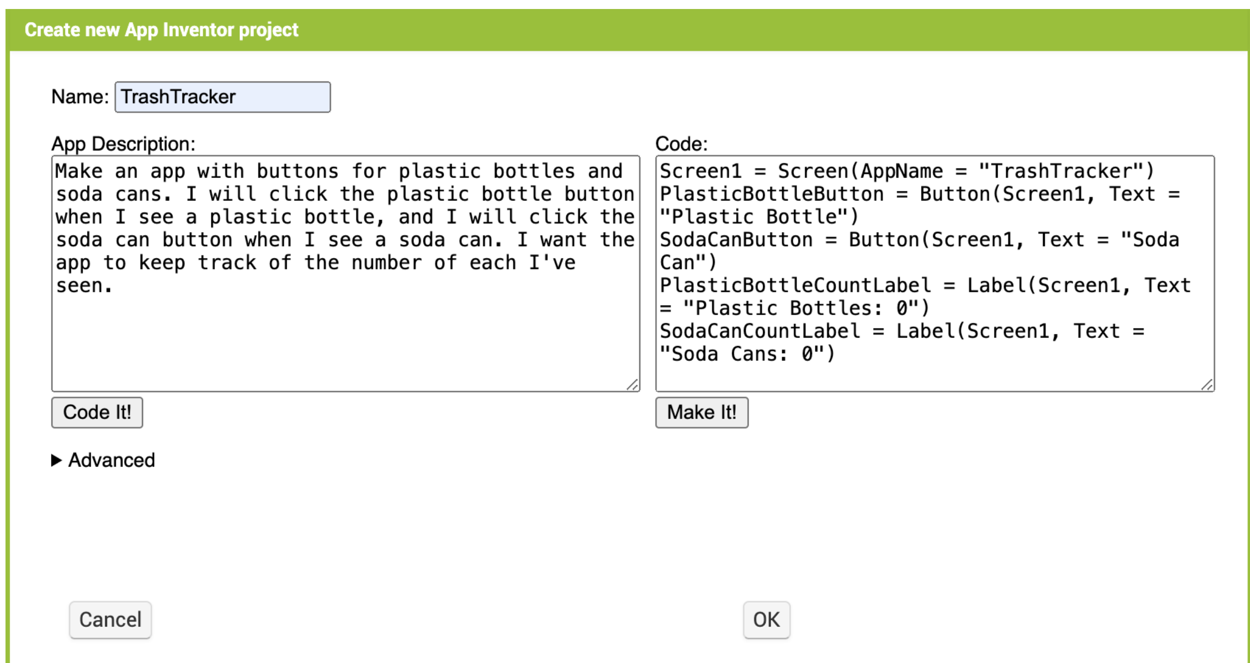


Figure 2.2: The app description and Aptly code for "TrashTracker" after the user has clicked "Code It!"

appear (figures 2.3 and 2.4).

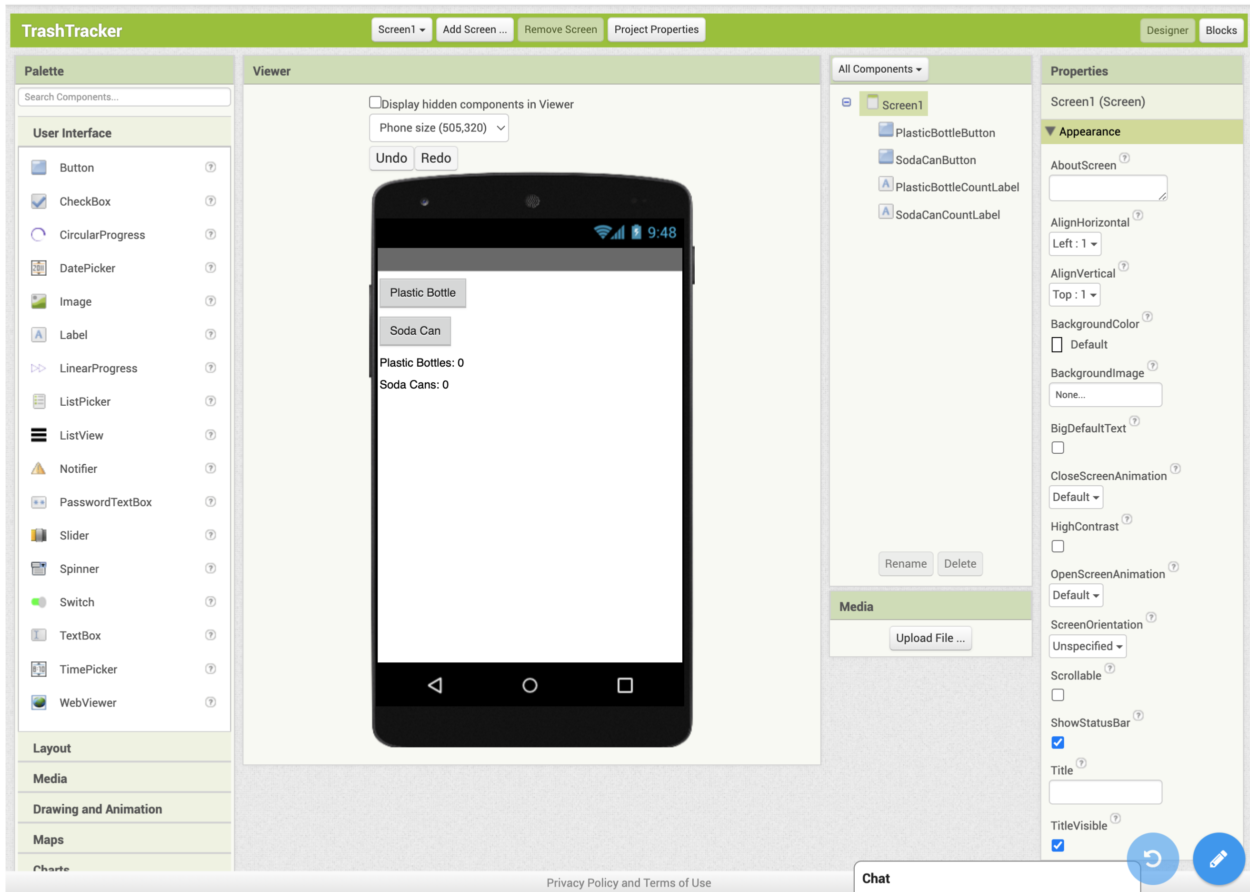


Figure 2.3: The mock screen for "TrashTracker" after the user has clicked "Make It!"

Abby loads the app onto her phone, and takes a stroll through her neighborhood using her app. She is able to keep track of the number of plastic bottles and metal soda cans, and the screen updates each time she clicks one of the buttons as expected, but she unfortunately notices many plastic bags in addition to plastic bottles and metal soda cans.

Back at her laptop, Abby decides to leverage Aptly Editing to adjust her app to help her keep track of plastic bags as well. She clicks the button with the pencil icon on the bottom right and instructs the AI assistant, "Add another button for plastic bags" as shown in figure 2.5.

Aptly Editing adds another button, as shown in figure 2.6, with corresponding code as shown in figure 2.7. Note that Abby did not explicitly state how this button should work,

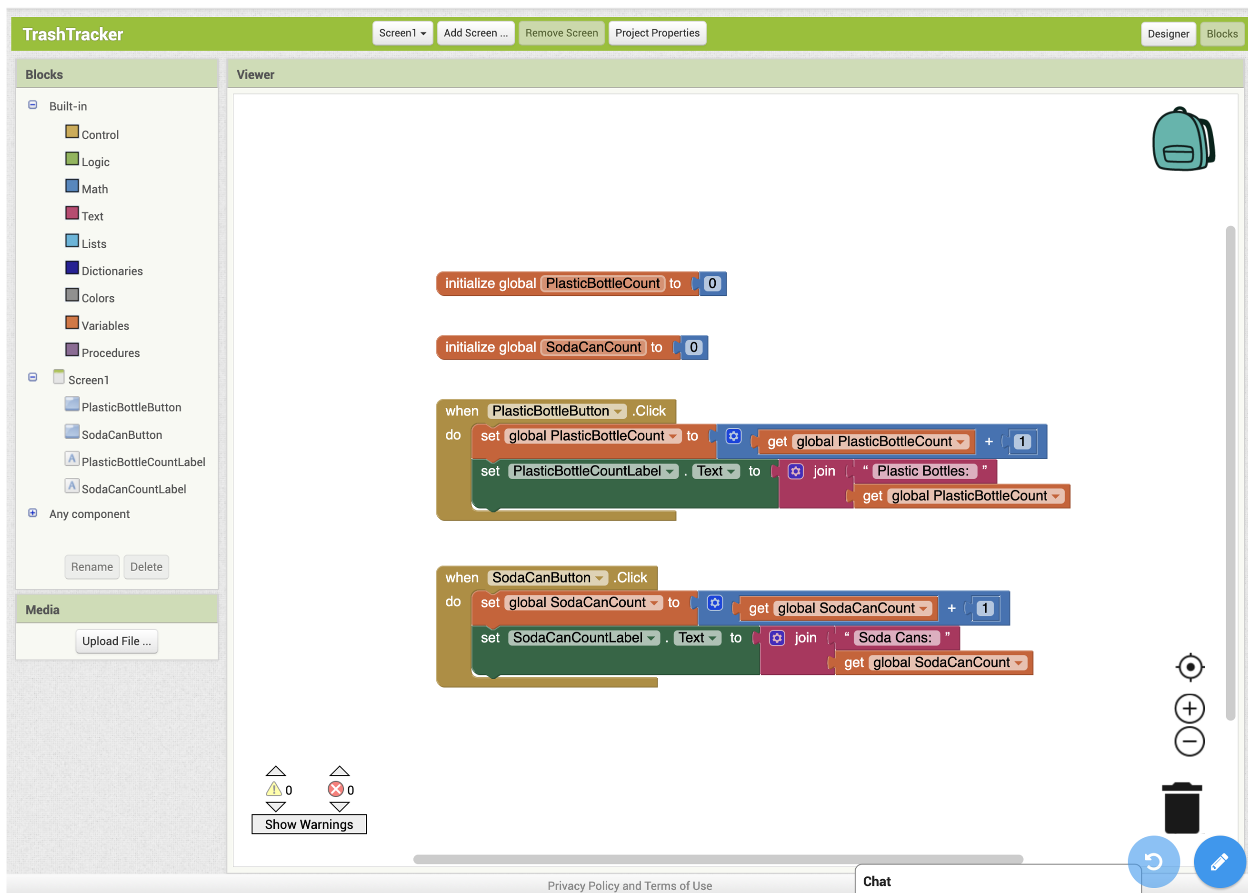


Figure 2.4: The blocks editor for “TrashTracker” after the user has clicked “Make It!”

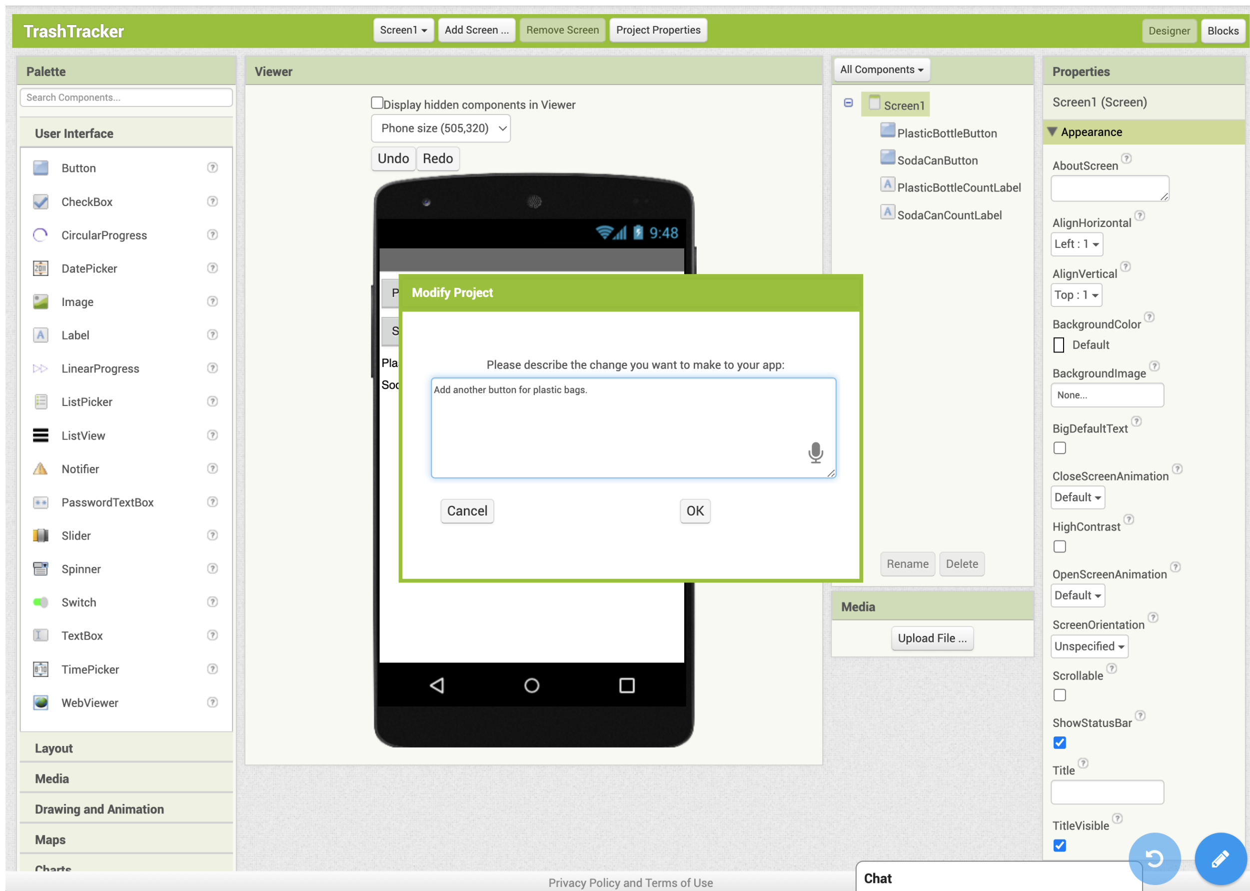

Figure 2.5: The edit description entered to help the student keep track of plastic bags.

but Aptly Editing infers that it should also add a label to display the number of times this new button has been pressed, so that the new button will function similarly to the existing buttons except for plastic bags rather than plastic bottles or soda cans.

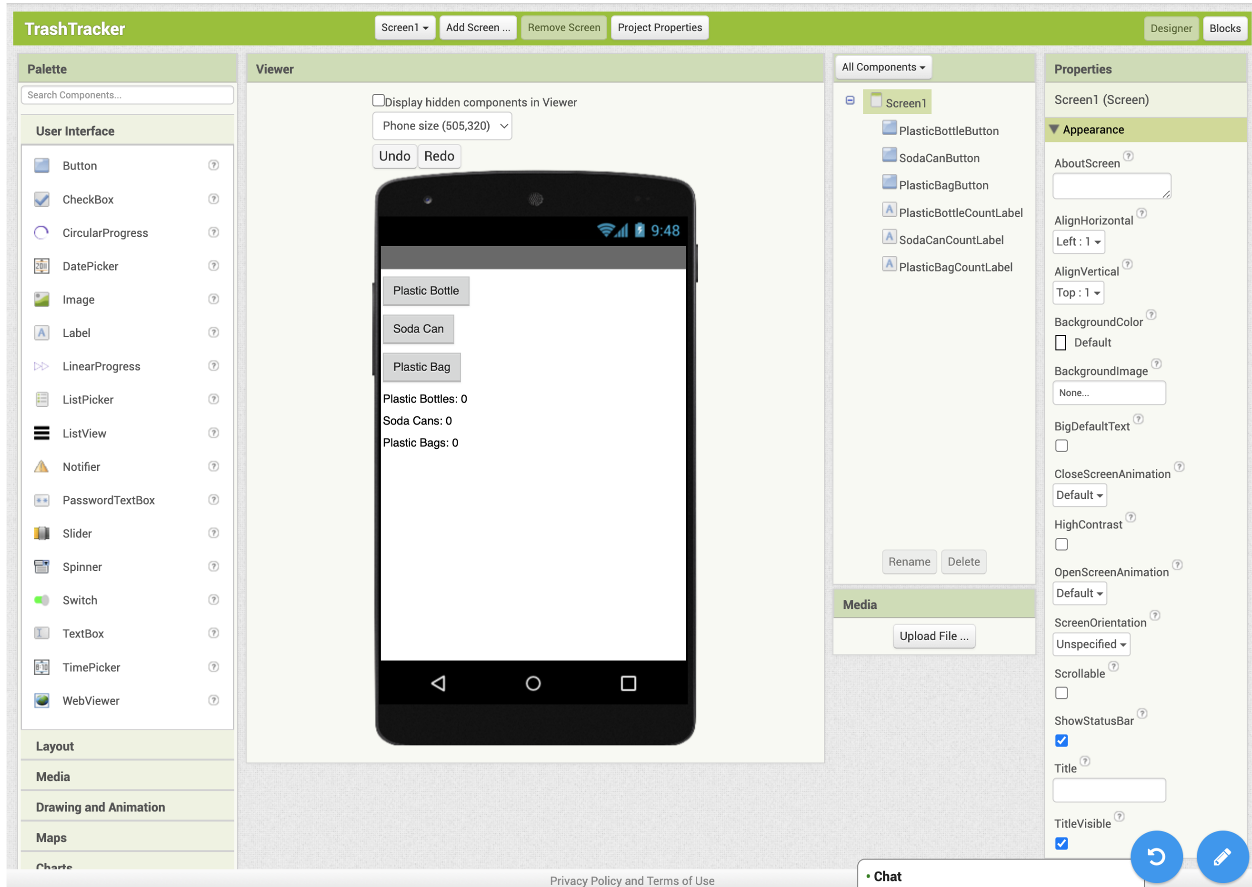


Figure 2.6: The updated mock screen with an additional button to help the student keep track of plastic bags.

Upon realizing that the task of walking through the neighborhood tapping the buttons will take longer than she originally anticipated, Abby enlists her friend Thomas for help. She wants Thomas to be able to use the app as well, and since he isn't familiar with the app, she wants the app to have instructions for its use. Abby instructs Aptly Editing, "Add instructions at the top of the screen so other people will know how to use the app" as shown in figure 2.8.

Aptly Editing adds a label that says "Click on the corresponding button each time you dispose of an item." (figure 2.9) This label text does not exactly match Abby's intention,

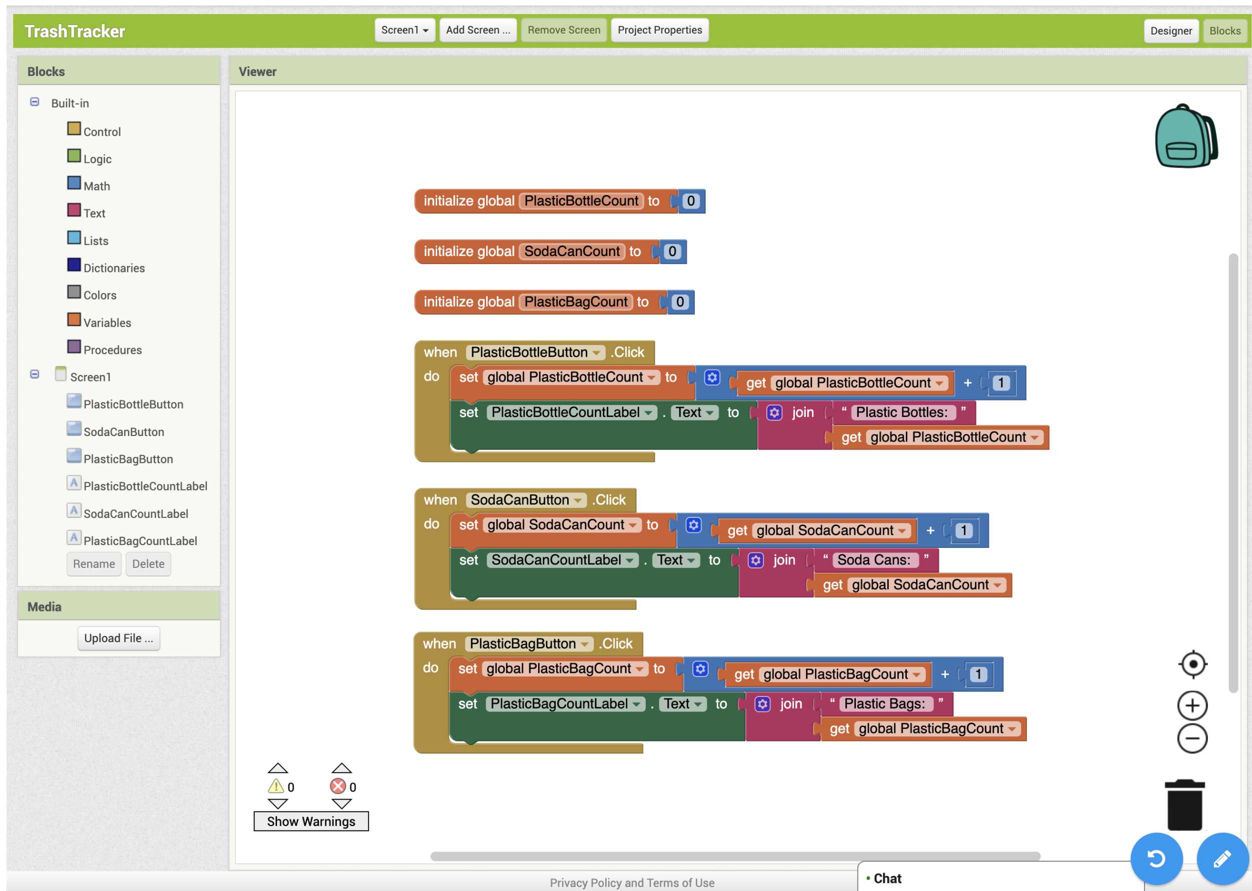


Figure 2.7: The updated blocks editor with code for an additional button to help the student keep track of plastic bags.

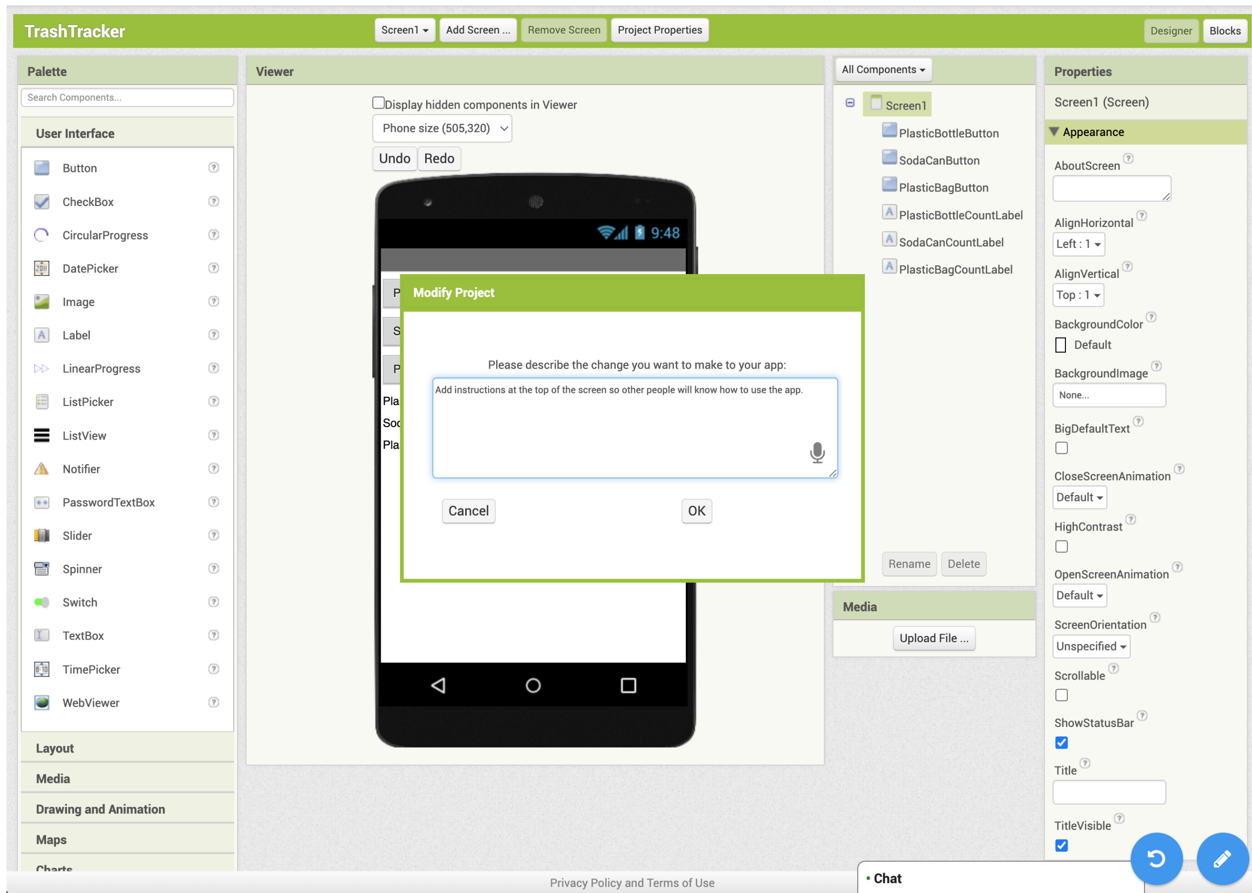


Figure 2.8: The student instructs Aptly Editing "Add instructions at the top of the screen so other people will know how to use the app."

since she is only tracking the litter and not planning on removing the trash herself or asking Thomas to do so. Additionally, the text is too large and is cut off on the mock screen. Abby decides to change the text and size of the label herself so the instructions will look exactly as she wants (figure 2.10). She tries out a few different label sizes until she gets one that fits the instructions perfectly on one line.

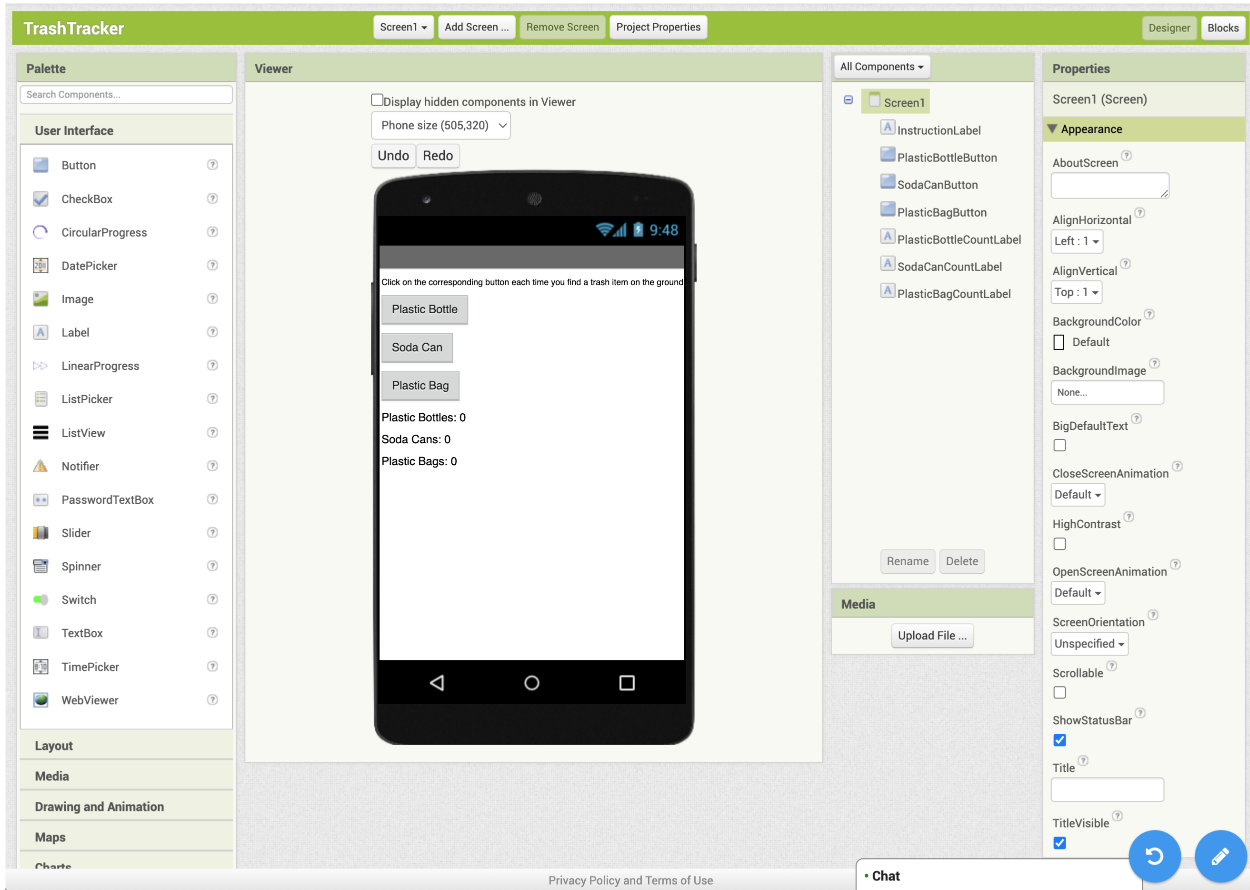


Figure 2.9: The result from the previous command.

Finally, Abby decides she wants the buttons and labels to be larger. She says, “Make everything bigger” (figure 2.11).

The result is what is shown in figure 2.12, which is not what Abby intended. She wanted the buttons and three bottom labels to be larger, but didn’t want the top label to become too wide for the screen. Abby decides to undo this change. She clicks the undo button at the bottom right and is presented with the screen shown in figure 2.13.

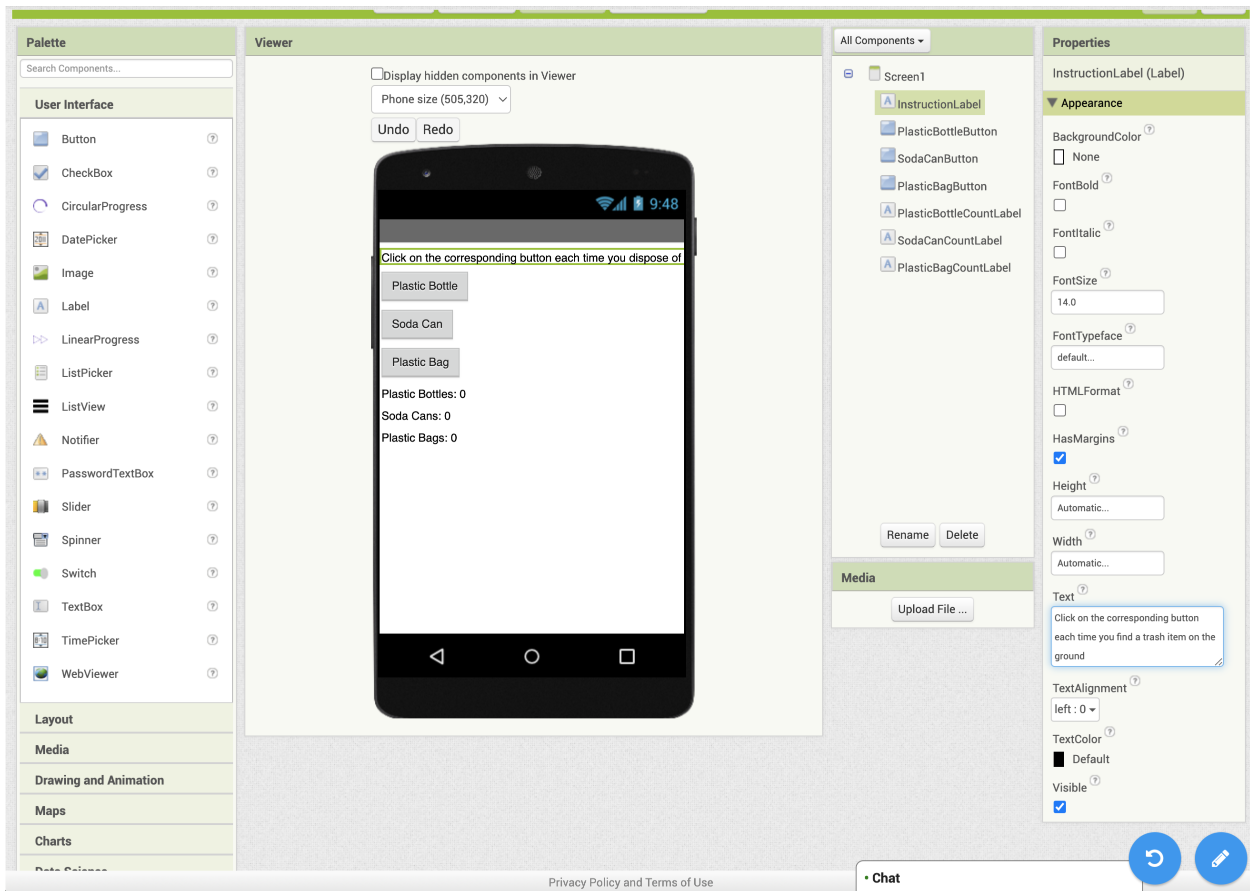


Figure 2.10: The student manually changing the label text.

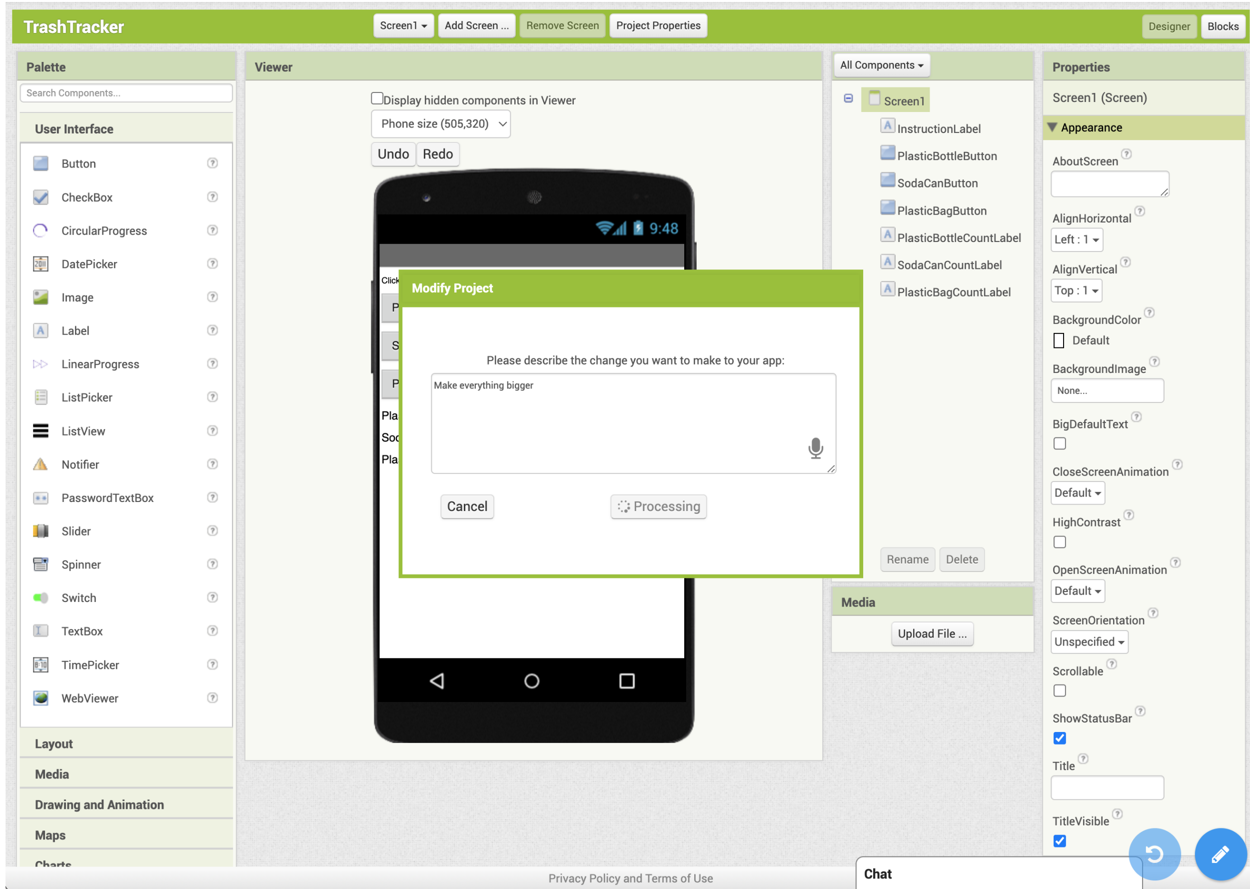


Figure 2.11: The command “Make everything bigger” is currently being processed by Aptly Editing.

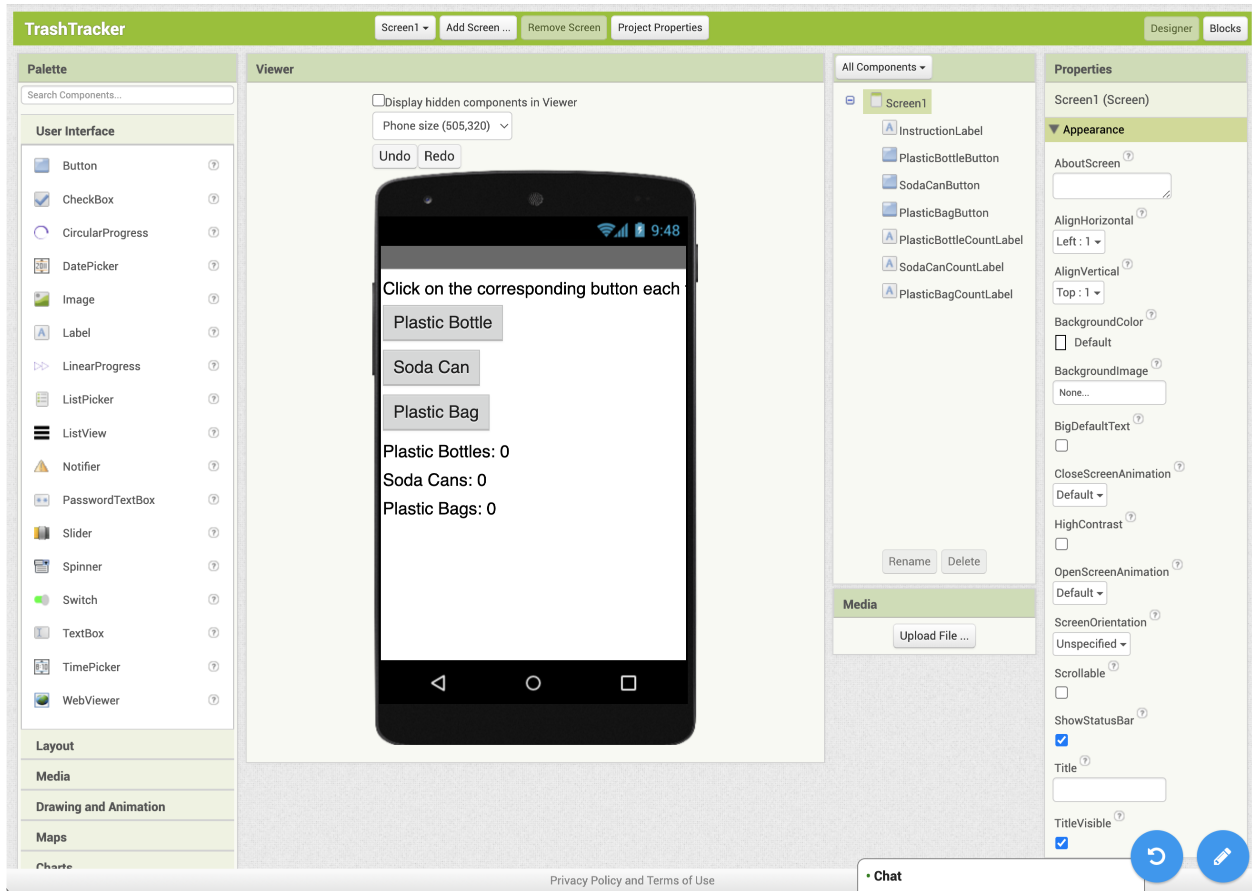


Figure 2.12: The result from the previous command, with the top label being cut off on the right.

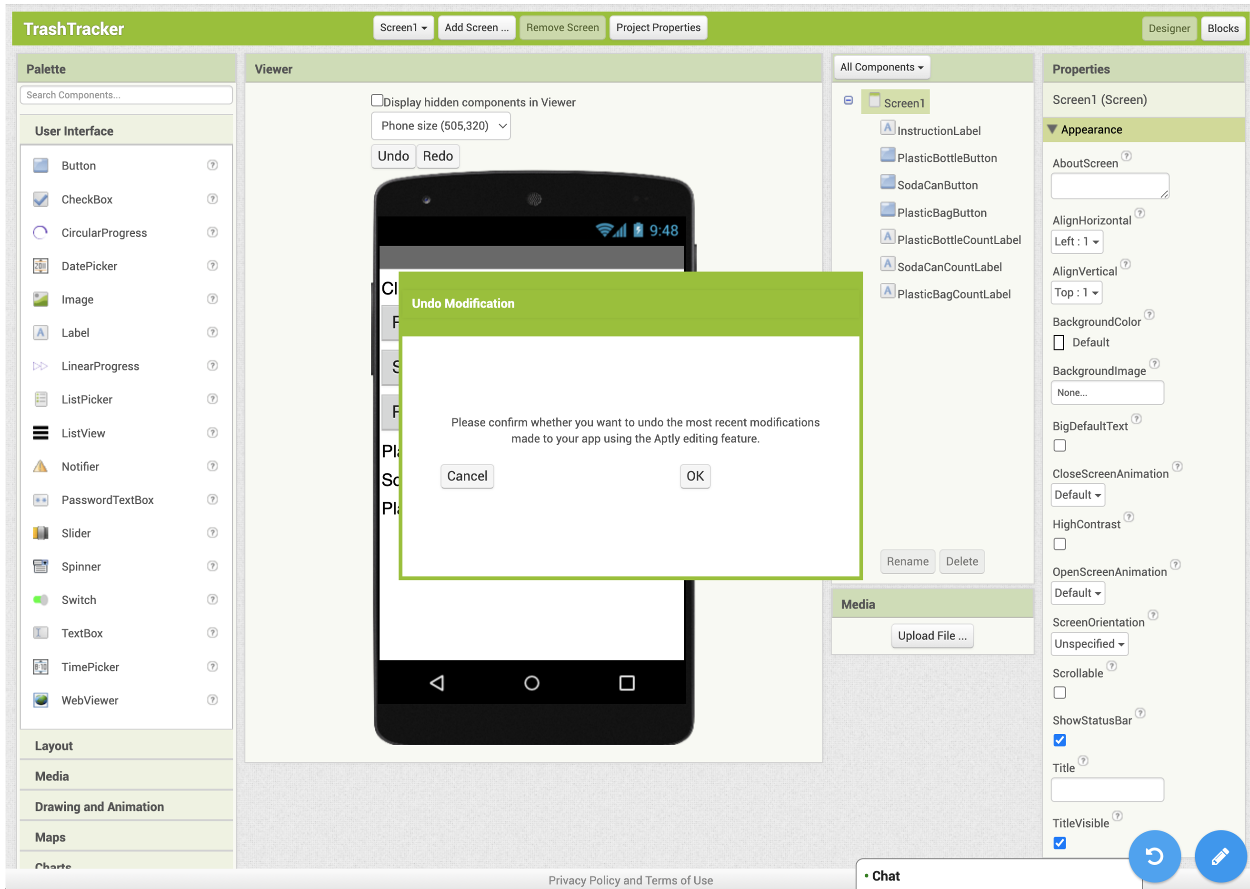


Figure 2.13: The student in the process of undo-ing Aptly Editing's change.

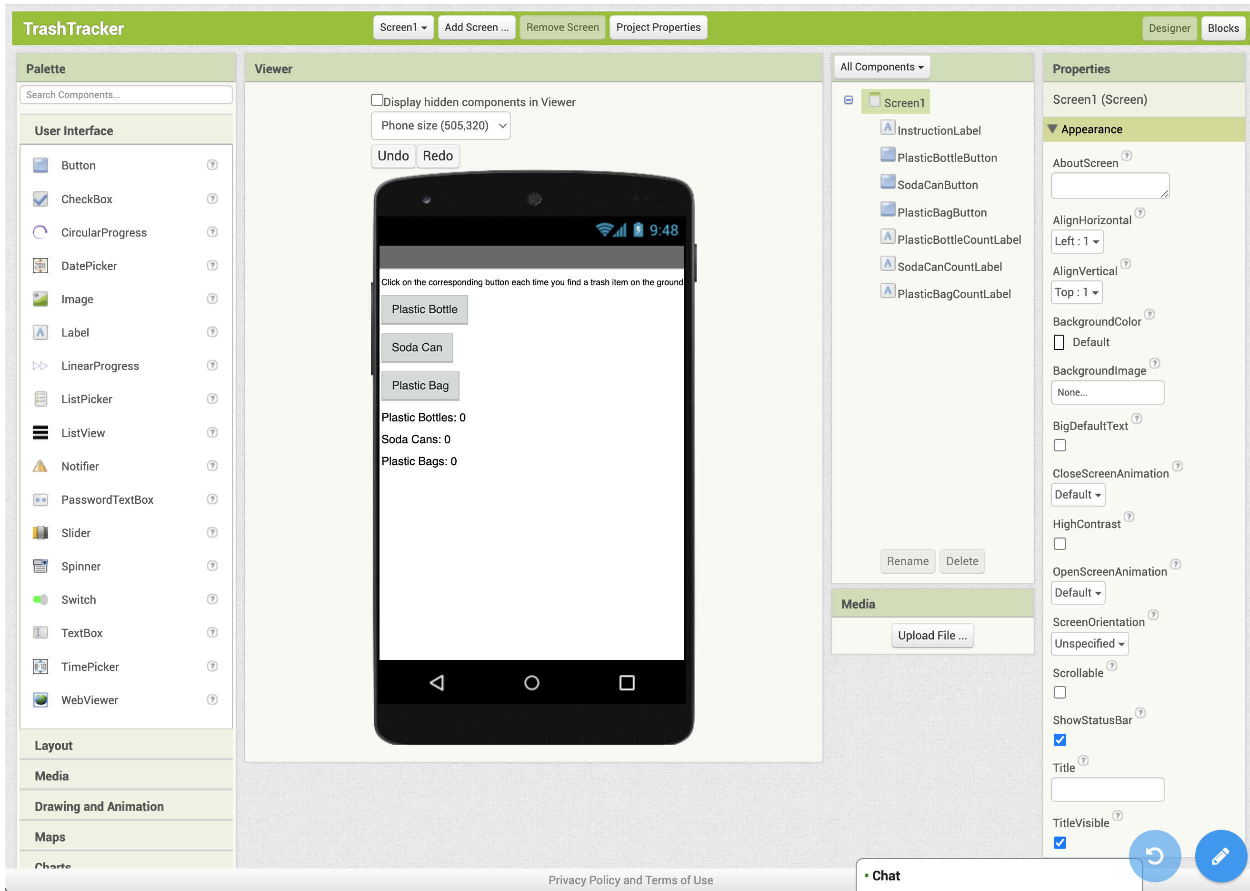


Figure 2.14: Aptly's edit has successfully been undone.

Aptly's edit has been successfully undone now as shown in figure 2.14, and Abby decides to leave the app as-is.

Satisfied with the app, Abby and Thomas set out in their neighborhood with the modified app loaded onto their mobile phones, and they track how many of each trash type they encounter.

## 2.2 High-level technical approach

At a high level, the technical implementation supporting this user interaction with the tool is as follows.

When Abby initially described the app to Aptly App Creation and clicked the "Code It!" button, her description along with examples from an example bank were sent to GPT-4 [9], which returned a text-based representation of an MIT App Inventor app. An MIT App Inventor app was then created from this text-based representation through a conversion process when Abby clicked the "Make it!" button.

When Abby described desired changes to her app to Aptly Editing, her edit description and the current app, along with the most relevant examples from an example bank, were sent to GPT-4, which returned a text-based representation of the modified app. Through comparisons between the Abstract Syntax Tree (AST) representation of the current and modified apps, a tree edit distance algorithm was used to calculate the most efficient way to transform the current app into the modified one. Events capable of being interpreted by App Inventor were created based on these operations and careful consideration of the individual components and code blocks involved. Finally, these events were emitted from the server running Aptly to a server running real-time collaboration software, and the edits were applied to the app to produce the desired result.

When Abby decided to undo a change made by Aptly Editing, all events in the accumulated undo stack that belong to the most recent change marked as performed by Aptly

Editing were run backwards (i.e. the opposite change was performed), effectively undoing the change.

# Chapter 3

# Methodology

## 3.1 Background regarding Aptly

### 3.1.1 Aptly code

The MIT App Inventor team has developed a Pythonic language called Aptly code with a 1:1 correspondence with MIT App Inventor. Every MIT App Inventor app can be uniquely represented in Aptly code and vice versa.

For instance, the initial Aptly code for the app presented in Chapter 2 was as follows:

```
Screen1 = Screen(AppName = "TrashTracker")
PlasticBottleButton = Button(Screen1, Text = "Plastic Bottle")
SodaCanButton = Button(Screen1, Text = "Soda Can")
PlasticBottleCountLabel = Label(Screen1, Text = "Plastic Bottles: 0")
SodaCanCountLabel = Label(Screen1, Text = "Soda Cans: 0")

initialize PlasticBottleCount = 0
initialize SodaCanCount = 0

when PlasticBottleButton.Click():
```

```
    set global PlasticBottleCount = global PlasticBottleCount + 1
    set PlasticBottleCountLabel.Text = text_join("Plastic Bottles: ",
        global PlasticBottleCount)

when SodaCanButton.Click():
    set global SodaCanCount = global SodaCanCount + 1
    set SodaCanCountLabel.Text = text_join("Soda Cans: ", global
       SodaCanCount)
```

Another example demonstrating a wider variety of components and functionality is as follows:

Here is a sample MIT App Inventor app (figures 3.1 and 3.2) and the corresponding Aptly code. This app allows users to select a shape, color, and size from dropdown menus. The user can then tap the screen to draw a shape in the specified color and size at the location where the user taps. There is also a button that the user can click to clear the canvas.

```
Screen1 = Screen(Title = 'Screen1')
Horiz1 = HorizontalArrangement(Screen1, AlignHorizontal = 3,
   AlignVertical = 2, Height = -1010, Width = -2)
ShapePicker = ListPicker(Horiz1, ElementsFromString = 'triangle,
   circle,square', Width = 100, Text = 'Shape')
ColorPicker = ListPicker(Horiz1, ElementsFromString = 'red, orange,
   yellow, green, blue, purple, pink', Width = 100, Text = 'Color')
SizePicker = ListPicker(Horiz1, ElementsFromString = '5, 10, 20, 50',
    Width = 100, Text = 'Size')
Canvas1 = Canvas(Screen1, Height = -2, Width = -2)
Horiz2 = HorizontalArrangement(Screen1, AlignHorizontal = 3,
   AlignVertical = 2, Height = -1010, Width = -2)
ClearButton = Button(Horiz2, Width = -1050, Text = 'Clear canvas')
```

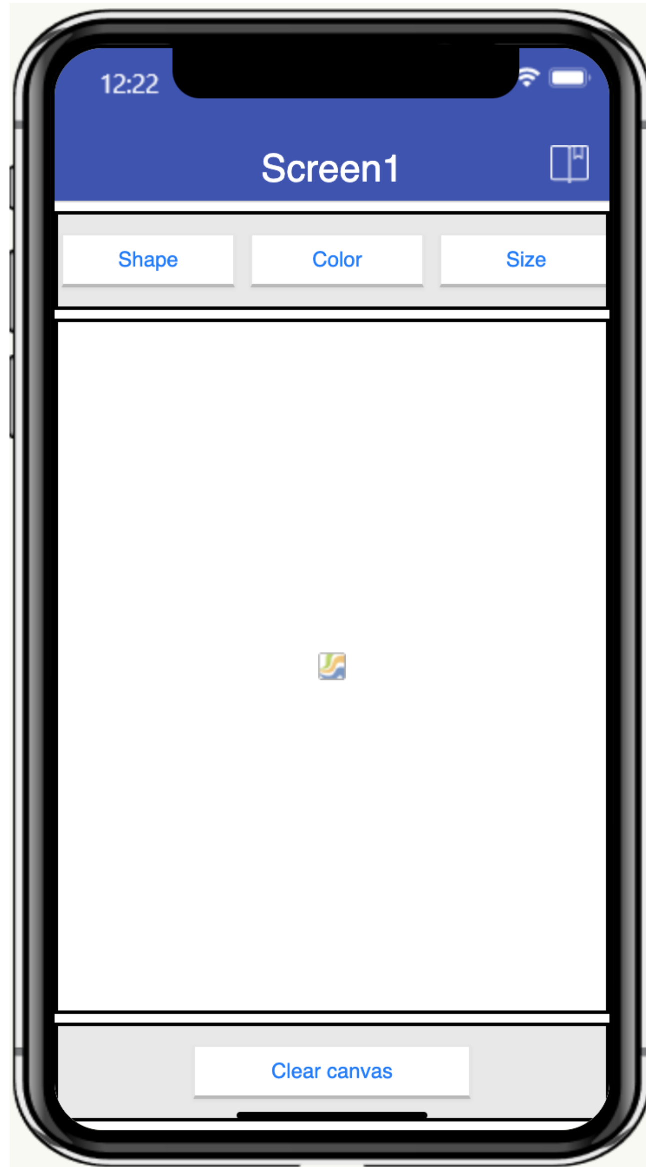


Figure 3.1: The mock screen for an app that allows users to select a shape, color, and size to draw in.

```
initialize global size to 5
initialize global shape to " square "
initialize global COLORS to make a dictionary
    key " red " value [red]
    key " orange " value [orange]
    key " yellow " value [yellow]
    key " green " value [green]
    key " blue " value [blue]
    key " purple " value [purple]
    key " pink " value [pink]

when ShapePicker .AfterPicking
do  set global shape to ShapePicker . Selection

when SizePicker .AfterPicking
do  set global size to SizePicker . Selection

when ColorPicker .AfterPicking
do  set Canvas1 . PaintColor to get value for key ColorPicker . Selection
                                 in dictionary get global COLORS
                                 or if not found " not found "

when ClearButton .Click
do  call Canvas1 .Clear

when Canvas1 .TouchUp
x y
do  if get global shape = " circle "
    then call Canvas1 .DrawCircle
            centerX get x
            centerY get y
            radius get global size / 2
            fill true
    else if get global shape = " triangle "
    then call Canvas1 .DrawShape
            pointList make a list
                          make a list get x
                                      get y + get global size
                          make a list get x + get global size / 2
                                      get y
                          make a list get x + get global size
                                      get y + get global size
            fill true
    else if get global shape = " square "
    then call Canvas1 .DrawShape
            pointList make a list
                          make a list get x
                                      get y
                          make a list get x + get global size
                                      get y
                          make a list get x + get global size
                                      get y + get global size
                          make a list get x
                                      get y + get global size
            fill true
```

Figure 3.2: The code blocks for an app that allows users to select a shape, color, and size to draw in.

```
initialize size = 5
initialize shape = 'square'
initialize COLORS = {'red': Color(0xFFFF0000), 'orange': Color(0
   xFFFFC800), 'yellow': Color(0xFFFFFF00), 'green': Color(0xFF00FF00
   ), 'blue': Color(0xFF3333FF), 'purple': Color(0xFF6600CC), 'pink':
    Color(0xFFFF99FF)}

when Canvas1.TouchUp(x, y):
  if global shape == 'circle':
    call Canvas1.DrawCircle(x, y, global size / 2, True)
  elif global shape == 'triangle':
    call Canvas1.DrawShape([[x, (y + global size)], [(x + global size
        / 2), y], [(x + global size), (y + global size)]], True)
  elif global shape == 'square':
    call Canvas1.DrawShape([[x, y], [(x + global size), y], [(x +
       global size), (y + global size)], [x, (y + global size)]],
       True)

when ShapePicker.AfterPicking():
  set global shape = ShapePicker.Selection

when SizePicker.AfterPicking():
  set global size = SizePicker.Selection

when ColorPicker.AfterPicking():
  set Canvas1.PaintColor = dictionaries_lookup(ColorPicker.Selection,
      global COLORS, 'not found')
```

```
when ClearButton.Click():
  call Canvas1.Clear()
```

This app contains several components: a `Canvas`, three `ListPicker`s (for selecting the shape, color, and size of the brush strokes), a `Button` for clearing the canvas, and two `HorizontalArrangement`s (for customizing the layout of the `ListPicker`s and `Button` on the screen).

The app uses variables (in orange) to hold the current size and shape (as selected via the `SizePicker` and `ShapePicker`, respectively), and it uses a dictionary `COLORS` (in dark blue) as a lookup table to map the string representation of colors (as selected via the `ColorPicker`) to the corresponding color value.

When the user taps the canvas (specifically when the user releases their press, in case the user is dragging their finger on the screen), a shape is drawn where the user released their press. The shape is created in the color selected (because the canvas paint color is set each time the user selects a new color via the `ColorPicker`) in the size selected. For drawing the shape, the `Canvas1.DrawCircle` procedure is used, but since there is no `DrawTriangle` method, for example, `DrawShape` is used for the triangle and square.

The Aptly representation of this app begins with the declaration of each component, where the type, name, parent component, and properties are specified.

Next, global variables are initialized.

Next, event handlers for events corresponding to the components are listed.

Although this app does not make use of such, user-defined procedures can also be utilized, and those definitions would be under the variable declarations as well.

### 3.1.2 Abstract Syntax Tree (AST) representation

MIT App Inventor apps can be represented not only in terms of Aptly code but also as an Abstract Syntax Tree (AST).

Nodes in the AST representation of an MIT App Inventor app can take one of dozens of

different types, such as `ComponentDecl`s for component declarations, `BoundComponentEvent`s for event handlers for events corresponding to components (such as `ListPicker.AfterPicking`), `SetFieldValue`s for variable assignments, `Identifier`s for component names such as `ColorPicker`, `ConditionalStatement`s for if/elseif/else statements, and `ConditionalBranch`es for individual if/else branches within a `ConditionalStatement`. These types are defined as Python classes and have a hierarchical structure, with `Node` being the base class.

Visual examples of ASTs are included later in this chapter.

#### Aptly code, AST, and .aia format conversion

The Aptly server contains several modules for converting between the Aptly code, AST, and .aia file representations of a given MIT App Inventor app:

- **Parser**: Produces an AST from Aptly code
- **Serializer**: Produces Aptly code from an AST
- **ProjectReader**: Produces Aptly code from an MIT App Inventor project file (.aia), and includes a parsing function to produce an AST as well
- **ProjectWriter**: Produces an MIT App Inventor project file from an AST

For example, an MIT App Inventor project can be converted into its AST representation via the ProjectReader which reads the .aia file, extracts the components and code blocks, and converts each component and block into a corresponding AST node or subtree.

## 3.2 Aptly Editing Implementation Details

### 3.2.1 User workflow for Aptly Editing

Given an in-progress MIT App Inventor app (created either by hand or by utilizing Aptly App Creation), Aptly Editing allows users to describe desired edits to that app in natural

language and have those edits be automatically implemented.

The user workflow for editing an in-progress app is as follows:

1. Click the blue pencil icon
2. Describe their desired edit
3. Watch as the edit is automatically implemented
4. Examine Aptly Editing's change, and if unhappy with the change, click the blue undo button
5. Repeat steps 1-4 with another edit description (if desired)

### 3.2.2 Editing Technical Process Overview

The editing procedure is comprised of the following steps (shown in figure 3.4):

1. The .aia file representing the current app and the edit description are sent from the server running App Inventor to the server running Aptly
2. The ProjectReader reads the .aia file and produces the AST representation of the current app
3. The Serializer takes this AST representation and produces Aptly code representing the original app
4. The Aptly server computes the embedding of the user description and the edit description for each example from the example bank, using OpenAI's Text/Code Embedding model (or retrieves the cached embedding if it has already been computed)
5. The Aptly server calculates the cosine similarity of the embedding of the user edit description with that of each example's edit description

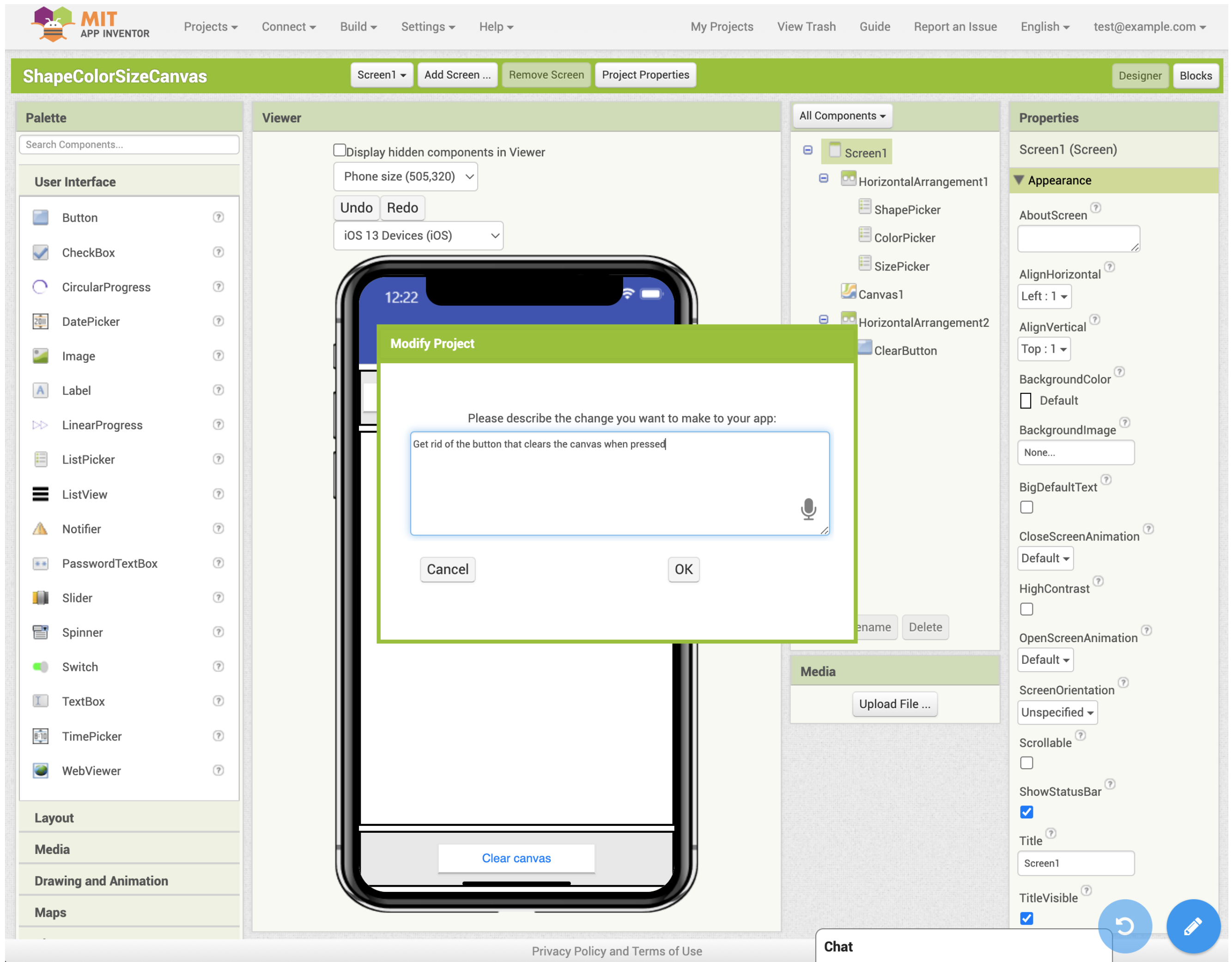


Figure 3.3: The Aptly Project Editor Wizard. In this example, the user has clicked the blue icon with the pencil icon and typed a description of their desired edit into the input text box. The undo button is disabled because the user has not yet made a change using Aptly.

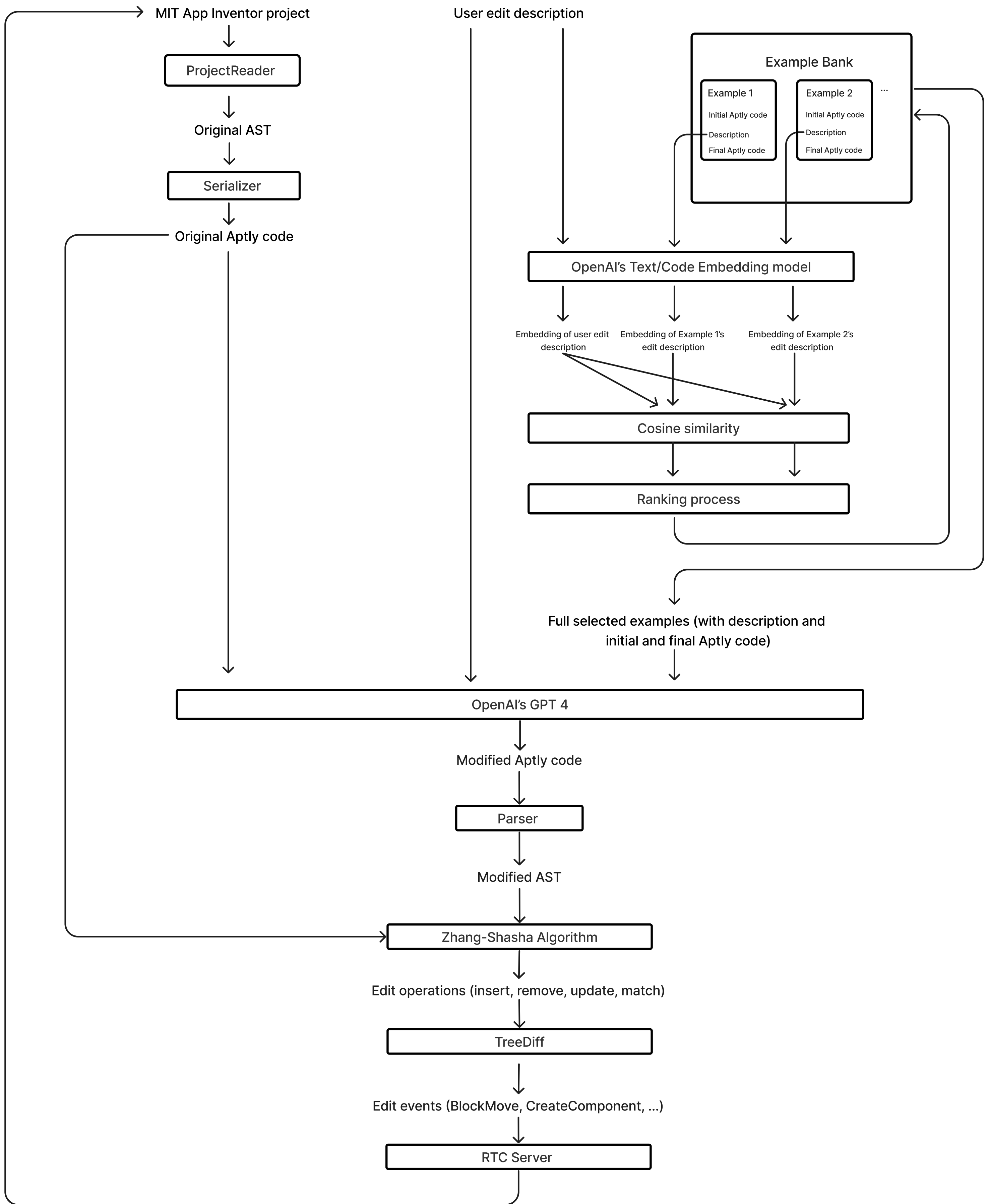

Figure 3.4: The editing technical process

6. The Aptly server selects relevant examples based on the highest cosine similarities as described in the previous step
7. The Aptly server sends the full example (the original Aptly code, edit description, and modified Aptly code for each selected example) along with the user's edit description and the Aptly code representing the original app to OpenAI's `gpt-4` model via its API, along with instructions to produce Aptly code representing the modified app
8. Using the Parser, the Aptly server produces the AST representation of the modified app
9. Using the Zhang-Shasha (ZSS) tree edit distance algorithm, the Aptly server computes the minimum (by cost) edit distance between the AST representing the original app and the AST representing the modified app, in terms of `Insert`, `Remove`, `Update`, and `Match` operations
10. Given the sequence of `Insert`, `Remove`, `Update`, and `Match` operations, the TreeDiff class computes a sequence of edit events in terms of changes to components and blocks (such as `BlockMove` and `CreateComponent`)
11. The Aptly server emits these events to the Real-Time Collaboration (RTC) server
12. The App Inventor server processes these RTC events and makes the changes to the project
13. If the user requests another edit, the process starts over from step 1

The following sections will go through the technical implementation of each of these steps.

### 3.2.3 The Pencil and Undo buttons

The pencil button for requesting edits as well as the undo button are implemented as buttons attached to the project editor. The click handler for the pencil button opens the Aptly

Project Editor Wizard (described in the next section), and the click handler for the undo button opens the Aptly Project Edit Undo Wizard.

### 3.2.4 Aptly Project Editor Wizard

The popup box for editing a project with Aptly is implemented as a Wizard (called AptlyProjectEditorWizard) in the version of the open-source `mit-cml/appinventor-sources` repository. When the "Ok" button is pressed, an XMLHttpRequest is sent to the Aptly server (which is an HTTP server written in Python using Flask) with the current project as a .aia file along with the user-provided edit description.

### 3.2.5 Generating Aptly code representing the current app

To generate Aptly code representing the current app, the ProjectReader converts the components and XML representation of the blocks and produces the corresponding AST nodes. Then, the Serializer converts this AST representation into Aptly code.

### 3.2.6 Ranking Examples

The example bank contains dozens of examples of edits, based on different types of edits users might make (adding components, removing components, changing properties, changing the functionality of components, changing the flow of logic in the app, etc.)

Each example takes the following form:

```
BEGININITIAL
Aptly code representing the full initial app here
ENDINITIAL

BEGINDESC
A natural language description of the desired change here
ENDDESC
```

```
BEGINFINAL
Aptly code representing the full final app here
ENDFINAL
```

Here is a simple sample example:

```
BEGININITIAL
Screen1 = Screen(Title = "Screen1")
Button1 = Button(Screen1, Text = "Click me!")

when Button1.Click():
    set Button1.Text = "You clicked me!"
ENDINITIAL

BEGINDESC
Add another button that does the same thing as Button1.
ENDDESC

BEGINFINAL
Screen1 = Screen(Title = "Screen1")
Button1 = Button(Screen1, Text = "Click me!")
Button2 = Button(Screen1, Text = "Click me!")

when Button1.Click():
    set Button1.Text = "You clicked me!"

when Button2.Click():
    set Button1.Text = "You clicked me!"
ENDFINAL
```

The text "BEGININITIAL", "ENDINITIAL", "BEGINDDESC", "ENDDESC", "BEGIN-FINAL", and "ENDFINAL" are used for easily extracting the initial code, edit description, and final code separately.

Due to limitations on the number of tokens that can be sent to OpenAI's `gpt-4` model, both in terms of a hard limit imposed by the API and also the per-token cost incurred, we want to send `gpt-4` a small number of relevant examples.

To determine the relevance of each example, the Aptly server utilizes OpenAI's Text/-Code Embedding model [10] to first compute the embedding of the edit description provided by the user, as well as the embedding of the edit description of each example. The Aptly server caches the computed embeddings in a JSON file and only recomputes them for a given example if that example changes, in order to minimize API calls and save on cost.

Given the embeddings of the user edit description and the edit descriptions of each example, the Aptly server calculates the cosine similarity between the embedding of the user's edit description and the embedding of each example's edit description and returns the top n examples based on the cosine similarity values. (n can be increased or decreased by changing a single variable in the code to send more or fewer examples to `gpt-4`).

### 3.2.7 Processing GPT-4's output

Given the Aptly code representing the modified project, the Parser converts the Aptly code into the AST representation.

### 3.2.8 Computing edit operations using Zhang-Shasha (ZSS) algorithm

The Zhang-Shasha (ZSS) algorithm [11] is a method for computing the minimum edit distance between two trees. ZSS utilizes dynamic programming and considers several types of edits, namely inserting, deleting, and updating nodes (in addition to matching nodes).

The Python ZSS module [12] is used to compute the minimum (by cost) sequence of edit operations (`Insert`, `Remove`, `Update`, and `Match`) needed to transform the AST representing the original app into the AST representing the modified app.

### Update Cost

The cost to `Insert` or `Remove` a node is always 1 and the cost to `Match` one node to another node is always 0. The cost to `Update` one node into another is not uniform for all node types, so the `update_cost` function takes into account the type of the two nodes for each proposed update.

If the two nodes are of different types, the update cost should be infinite since it is not possible to transform a node of one type into a node of another type.

Even if the two nodes are of the same type, the update cost can still be infinite. For example, a `ComponentDecl` declaring a Button cannot be transformed into a `ComponentDecl` declaring a Label, because Buttons and Labels are different types that cannot be converted into each other within App Inventor. A user who currently has a Button on the screen and wants a Label instead would have to select a Label from the palette and delete the Button, so Aptly also cannot change a Button into a Label.

Another example of infinite cost is two `BoundComponentEvent`s where the corresponding components are different types (such as `Button1.GotFocus` vs. `Switch1.GotFocus`) or the method name is different (such as `Button1.GotFocus` vs `Button1.Click`), because in both of these cases, there is no way, in App Inventor, to make such a transformation. A user trying to make the former change would have to click on Switch1 and select the `Switch1.GotFocus` option from the blocks drawer, or select `Switch1` in the dropdown of switches on any `Switch.GotFocus` method, and then remove the `Button1.GotFocus` if desired). A user trying to make the latter change would also have to select `Button1.Click` from the blocks drawer (and delete the `Button1.GotFocus` if desired).

Other types of transformations have finite cost. For example, for primitive types such

as NumberValue, StringValue, and BooleanValue, the update cost is 1 if the values are different (such as 5 vs. 6 for a NumberValue) or 0 otherwise.

**Example edit operation sequence: Changing button text**

Suppose the user has an app with a single button, corresponding to this initial code:

```
Screen1 = Screen(Title = 'Screen1')
Button1 = Button(Screen1, Text = 'Text for Button1')
```

If the user says "Change the button text to 'Click me!"' then the resulting Aptly code might look like this:

```
Screen1 = Screen(Title = 'Screen1')
Button1 = Button(Screen1, Text = 'Click me!')
```

In this case, figure 3.5 shows what the change will look like at the AST level:

And here are the edit operations produced by the ZSS algorithm:

```
<Operation Match> Screen1 Screen1 (Identifier)
<Operation Match> Screen1 Screen1 (Reference)
<Operation Match> Screen Screen (Identifier)
<Operation Match> Title Title (Identifier)
<Operation Match> Title Title (Reference)
<Operation Match> Screen1 Screen1 (StringValue)
<Operation Match> Title = Screen1 Title = Screen1 (DesignerAssignment
   )
<Operation Match> ComponentTypeInvocation(Identifier('Screen'), None,
    {Identifier('ActionBar'): BooleanValue(True), Identifier('AppName
   '): StringValue('OneButton'), Identifier('Title'): StringValue('
   Screen1')}) ComponentTypeInvocation(Identifier('Screen'), None, {
   Identifier('ActionBar'): BooleanValue(True), Identifier('AppName')
   : StringValue('OneButton'), Identifier('Title'): StringValue('
```

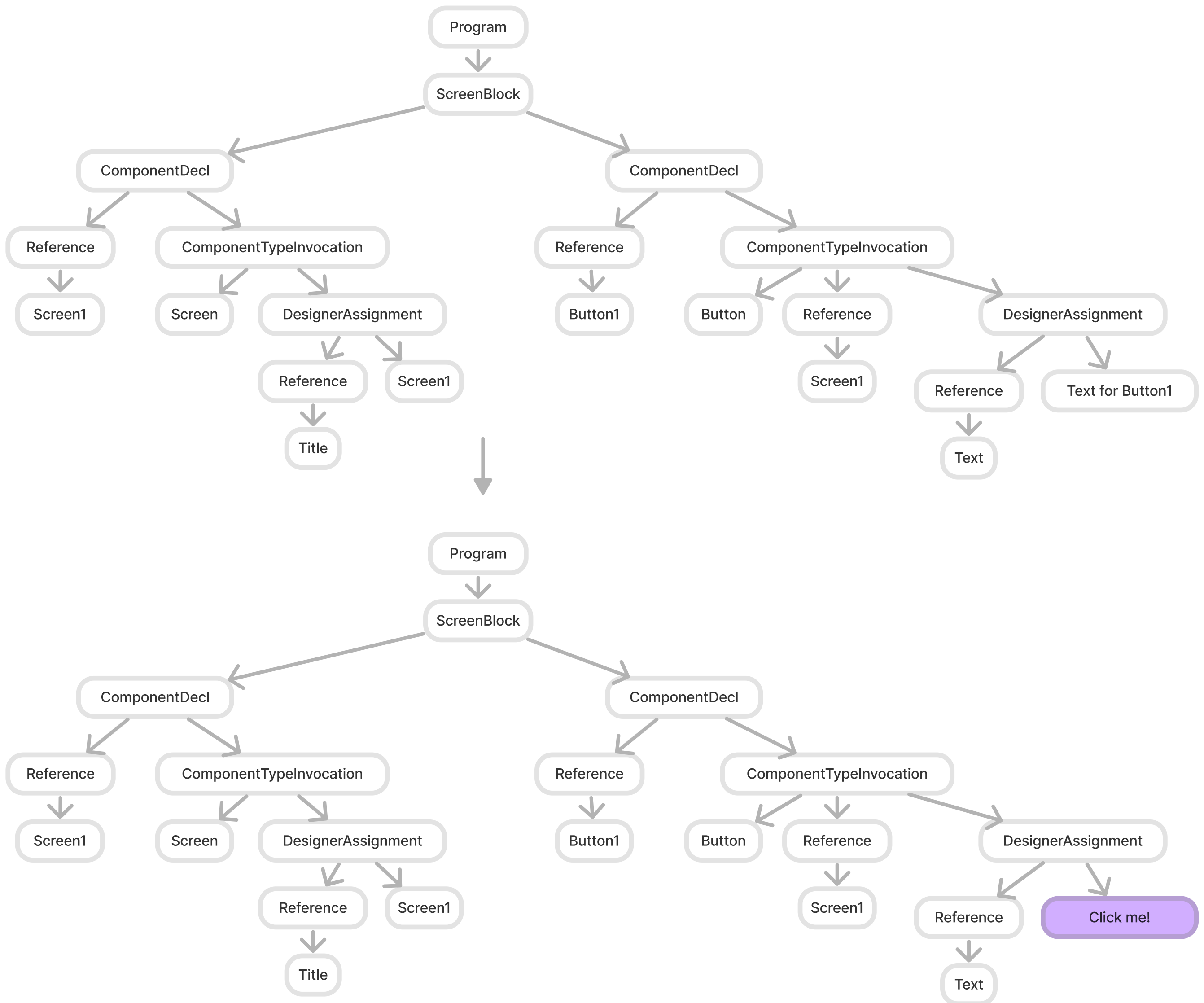


Figure 3.5: The button text change reflected at the AST level. Matched nodes are shown in white and Updated nodes are shown in purple.

```
    Screen1')})
<Operation Match> Screen1 = ComponentTypeInvocation(Identifier('
    Screen'), None, {Identifier('ActionBar'): BooleanValue(True),
    Identifier('AppName'): StringValue('OneButton'), Identifier('Title
    '): StringValue('Screen1')}) Screen1 = ComponentTypeInvocation(
    Identifier('Screen'), None, {Identifier('ActionBar'): BooleanValue
    (True), Identifier('AppName'): StringValue('OneButton'),
    Identifier('Title'): StringValue('Screen1')}) (ComponentDecl)
<Operation Match> Button1 Button1 (Identifier)
<Operation Match> Button1 Button1 (Reference)
<Operation Match> Button Button (Identifier)
<Operation Match> Screen1 Screen1 (Identifier)
<Operation Match> Screen1 Screen1 (Reference)
<Operation Match> Text Text (Identifier)
<Operation Match> Text Text (Reference)
<Operation Update> Text for Button1 Click me! (StringValue)
<Operation Match> Text = Text for Button1 Text = Click me! (<class '
    aptly.ast.DesignerAssignment'>)
<Operation Match> ComponentTypeInvocation(Identifier('Button'),
    Reference(Identifier('Screen1')), {Identifier('Text'): StringValue
    ('Text for Button1')}) ComponentTypeInvocation(Identifier('Button
    '), Reference(Identifier('Screen1')), {Identifier('Text'):
    StringValue('Click me!')})
<Operation Match> Button1 = ComponentTypeInvocation(Identifier('
    Button'), Reference(Identifier('Screen1')), {Identifier('Text'):
    StringValue('Text for Button1')}) Button1 =
    ComponentTypeInvocation(Identifier('Button'), Reference(Identifier
    ('Screen1')), {Identifier('Text'): StringValue('Click me!')}) (
    ComponentDecl)
```

```
<Operation Match> ScreenBlock(ComponentDecl(Reference(Identifier('
    Screen1')), ComponentTypeInvocation(Identifier('Screen'), None, {
    Identifier('Title'): StringValue('Screen1')})), ComponentDecl(
    Reference(Identifier('Button1')), ComponentTypeInvocation(
    Identifier('Button'), Reference(Identifier('Screen1')), {
    Identifier('Text'): StringValue('Text for Button1')})))
    ScreenBlock(ComponentDecl(Reference(Identifier('Screen1')),
    ComponentTypeInvocation(Identifier('Screen'), None, {Identifier('
    Title'): StringValue('Screen1')})), ComponentDecl(Reference(
    Identifier('Button1')), ComponentTypeInvocation(Identifier('Button
    '), Reference(Identifier('Screen1')), {Identifier('Text'):
    StringValue('Click me!')})))
```

**Example edit operation sequence: Changing a bound component event**

Suppose the user has an app with a single button, corresponding to this initial code:

```
Screen1 = Screen(Title = 'Screen1')
Button1 = Button(Screen1, Text = 'Text for Button1')

when Button1.Click():
  set Button1.BackgroundColor = Color(0xFFFF0000)
```

If the user says "When the button is clicked, make the button blue instead of red" then the resulting Aptly code might look like this:

```
Screen1 = Screen(Title = 'Screen1')
Button1 = Button(Screen1, Text = 'Text for Button1')

when Button1.Click():
  set Button1.BackgroundColor = Color(0xFF0000FF)
```

In this case, here is what the change will look like at the AST level (note that some unchanged nodes are omitted for readability):

And here are the edit operations produced by the ZSS algorithm (note that match operations for the screen are omitted for brevity):

```
(Match operations for the ScreenBlock and all nodes that are part of
   that subtree)
<Operation Match> Button1 Button1 (Identifier)
<Operation Match> Click Click (Identifier)
<Operation Match> Button1 Button1 (Identifier)
<Operation Match> BackgroundColor BackgroundColor (Identifier)
<Operation Match> Button1.BackgroundColor Button1.BackgroundColor (
   ComponentFieldName)
<Operation Update> ColorValue('0xFFFF0000') ColorValue('0xFF0000FF')
<Operation Match> Button1.BackgroundColor = ColorValue('0xFFFF0000')
   Button1.BackgroundColor = ColorValue('0xFF0000FF') (SetFieldValue)
<Operation Match> BoundComponentEvent(Identifier('Button1'),
   Identifier('Click')) BoundComponentEvent(Identifier('Button1'),
   Identifier('Click')) (BoundComponentEvent)
<Operation Match> Program(ScreenBlock(ComponentDecl(Reference(
   Identifier('Screen1')), ComponentTypeInvocation(Identifier('Screen
   '), None, {Identifier('Title'): StringValue('Screen1')})),
   ComponentDecl(Reference(Identifier('Button1')),
   ComponentTypeInvocation(Identifier('Button'), Reference(Identifier
   ('Screen1')), {Identifier('Text'): StringValue('Text for Button1')
   }))), [BoundComponentEvent(Identifier('Button1'), Identifier('
   Click'))]) Program(ScreenBlock(ComponentDecl(Reference(Identifier
   ('Screen1')), ComponentTypeInvocation(Identifier('Screen'), None,
   {Identifier('Title'): StringValue('Screen1')})), ComponentDecl(
```

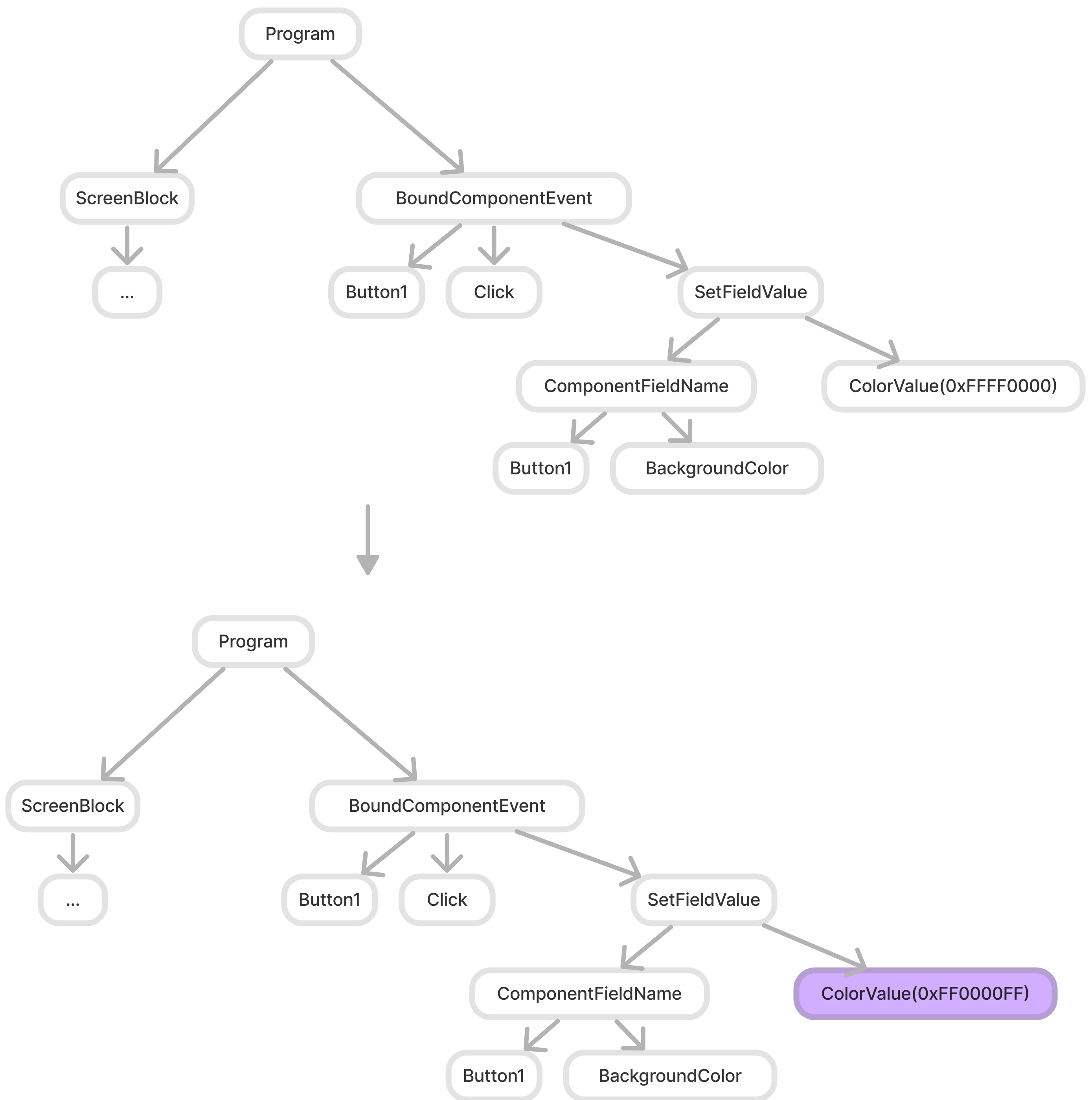


Figure 3.6: The bound component event change reflected at the AST level. Matched nodes are shown in white and Updated nodes are shown in purple.

```
Reference(Identifier('Button1')), ComponentTypeInvocation(
Identifier('Button'), Reference(Identifier('Screen1')), {
Identifier('Text'): StringValue('Text for Button1')}))), [
BoundComponentEvent(Identifier('Button1'), Identifier('Click'))])
```

Examples with insertions and deletions are shown in the following section.

### 3.2.9 Creating edit events

Given the sequence of operations returned from ZSS, it is necessary to process these operations and emit a sequence of edit events capable of being interpreted by App Inventor.

Directly emitting a change such as `<Operation Update> NumberValue(1) NumberValue(2)` would not result in the appropriate change being made in App Inventor, because while there are blocks representing number values in the Blocks editor and number-based inputs such as font size in the Designer editor, the `ast.NumberValue` type does not uniquely identify a type of App Inventor block or property and `ast.NumberValue` is a type used entirely within the Aptly server; there is nothing in App Inventor capable of understanding the needed change.

A change such as changing `when Button1.Click():` to `when Button2.Click():` would have the edit operation be updating the `ast.Identifier` `Button1` to `Button2` and matching every other node including the `BoundComponentEvent`. However, the way a change like this gets applied in App Inventor is changing the `COMPONENT_SELECTOR` of the component event block from `Button1` to `Button2`.

Additionally, changes such as inserting or removing a list item or dictionary pair, or inserting or removing an item at the beginning or middle of a stack of blocks, can require issuing multiple `BlockMove` events to make sure each code block's parent pointer and input name are correct.

For these reasons (along with many analogous reasons for other types of changes), it is necessary to compute a sequence of edit events to emit to App Inventor that make use of App Inventor's block and component types rather than the AST node types, and that are

often more extensive than the operations returned from ZSS.

Events are emitted in JSON format and constructed using Python classes which set up JSONs in the format corresponding to that event type in the App Inventor server in JavaScript. The events are superclasses of either `ComponentEvent` for events involving the components and `BlockEvent` for events involving the code blocks. Both `ComponentEvent` and `BlockEvent` are superclasses of the `Event` class which contains fields for the `type` and `group`. (Groups are used when multiple events are issued at the same time for part of the same edit, so that those events can also be undone together at the click of an undo button.)

The types of `ComponentEvent`s that can be emitted are as follows:

- `ComponentAdd`, with type `component.create` which takes in a `componentType` indicating the type of the component (such as Button or Label), `component` which is a dictionary with the component's properties, and `parentId` and `index` for proper ordering on the screen within its parent (which may be the screen itself or some type of layout arrangement).

- `ComponentRemove`, with type `component.delete` which takes in the `component` dictionary as well as `parentId` and `index` for being able to restore the component to the correct place if this event is undone. This event type also contains an `ids` field which includes the component's ID as well as that of any child components because deleting a component will also delete its children in MIT App Inventor.

- `ComponentProperty`, with type `component.property`, which takes in the name of the property as well as the property's current value and the new value.

- `ComponentRename`, which is a superclass of `ComponentProperty` and which changes the `Name` property to be the new value and also copies over the `componentId` from the original component.

- `ComponentProperty`, with type `component.property`, which takes in the name of the property as well as the property's current value and the new value.

- `ComponentMove` with type `component.move` which takes in the new and old parent ID and parent index

The types of `BlockEvents` that can be emitted are as follows:

- `BlockCreate`, with type `create`, which takes in the id for the block (`blockId`), the ids of all blocks that are children being created (`ids`), and the `xml` representing the blocks to be created.
- `BlockDelete`, with type `delete`, which takes in the id for the block (`blockId`) and the ids of all blocks that are children being deleted (`ids`), since removing a block removes its children in MIT App Inventor.
- `BlockChange`, with type `change`, which takes in the `blockID`, the `name` of the block, the `element` being changed (such as "mutation"), and the new and old value for that element.
- `BlockMove`, with type `move`, which takes in the id of the block and new parent, as well as the new input name and coordinate of the block. If `newParentId`, `newInputName`, and `newCoordinate` are all not provided, the block is moved out of any parent and socket it may currently be placed in.

**Process overview**

Given the sequence of edit operations, here is the rough order of processing steps:

1. Copy all of the operations into dictionaries for insertions, removals, updates, and matches, keyed by the id of the relevant node.
    - For each insertion, "preinsert" the node, meaning add the node to the dictionary of inserted blocks or inserted components (depending on the type).
2. Walk the original AST in breadth-first order to process removals, updates, and matches.

- For removals: Check if the same node is in the dictionary of inserted components or blocks, and if so, perform a move (`BlockMove` or `ComponentMove`) rather than a removal and then insertion. If performing a removal (`BlockDelete` or `ComponentRemove`), call the update function for the parent node (using `__update_parent__`) in order to adjust the parent pointers and input names for all children of the parent node.
- For matches: Copy over the `aiatools_ref`, which is a dictionary containing information about the node's mutation, input name, and ID, from the node in the original AST to the corresponding node in the modified AST.
- For updates: Update the node. This likely means changing a single field for primitive types, and may be a more complicated change for complex types.

3. Log the procedure-to-argument name mappings for all procedures, in order to be able to process updates to procedures later.
4. Walk the modified AST in breadth-first order to process insertions. For each insertion, call the update function for the parent node in order to adjust the parent pointers and input names for all children of the parent node.

**Designer Removals**

For removing components, a `ComponentRemove` event is emitted. Additional logic is added to the App Inventor server to handle taking the given event and attaching the corresponding block IDs to the event before running it so all blocks associated with the component being removed are also removed.

For removing `DesignerAssignments`, a `ComponentProperty` event is emitted changing the specified property from its current value to the default value. The default value is obtained from a lookup table populated from a JSON file.

### Block Removals

For block removals, the `BlockDelete` event is emitted with the IDs of the block being removed as well as all of its descendants. Blocks are also moved out of their current parent and socket if applicable before they are deleted, using the `BlockMove` event.

For more complicated types like `ConditionalBranch`es, more complex logic is needed to remove only the parts of that node that correspond to the blocks that need to be removed, or to avoid deleting blocks if we're removing a `ast.PassStatement` or `ast.NoneValue` which doesn't correspond to any block in MIT App Inventor.

Once a block is removed, it is necessary to call the update function on the parent node to adjust its inputs or stack body as needed.

### Designer Updates

For renaming components, a `ComponentRename` event is emitted with the new name.

Updates to `DesignerAssignment`s will not happen on their own, but the update function on a `DesignerAssignment` may be called by one of its children. In the update function for designer assignments, we find the match operation corresponding to the `DesignerAssignment` in order to obtain the relevant ID and emit a `ComponentProperty` event containing the old and new values for the specified property.

### Block Updates

The logic for updating blocks is more complicated and requires many special cases to the point where there are separate update functions defined for each node type.

Two common types of updates requiring complicated logic are insertions or removals in the middle of a stack body or a list of items.

Examples of list items are lists (where each element is an item) and dictionaries (where each element is a key/value pair). When an item is inserted into or removed from a list of items, the update function is called on the parent node. The update function of the

parent node calls a special `update_items` function, which first updates the mutation if the number of items has changed and the mutation has not been updated already, and then goes through each socket of the modified list or dictionary and checks if the correct element is already in place. If so, it does nothing. If not, it finds the correct item (which may be in an update, match, or insert operation) and moves it to the correct place. Moving a block to the correct place means updating its parent ID if needed and updating its input name to `ITEMi` or whatever the input name is called for that particular component type. (Note that for insertions, if the block has not been set up yet, it will be placed in the right socket once it is created since the update parent function will get called again when it is created. To facilitate this process, we keep track of which blocks we have already moved to avoid issuing duplicate events.)

Examples of stack bodies are the body of a component event, procedure, the "then" body in a conditional statement, etc. When a statement is inserted into or removed from a stack, the update function is called on the parent node. The update function of the parent node calls a special `update_stack_body` function, which goes through each socket of the modified stack and checks if the correct element is already in place. If so, it does nothing. If not, it finds the correct item (which may be in an update, match, or insert operation) and moves it to the correct place. Since stacks of statements are represented as linked lists in App Inventor, moving a block to the correct place means updating its parent ID to the ID of its predecessor, not just to the ID of the event block, control block, or procedure block it is part (of unless it is the first element in the stack), in addition to updating its input name to `STACKi` or whatever the input name is called for that particular component type. (Like list item updates, items are moved to the correct place once they are created, so update may be called on the parent multiple times).

Here are some examples of update functions for different types of nodes:

- `BoundComponentEvent`s:
  - This type can only be updated by a child, so find the match operation to gain

access to the ID.

  - Handle changes and reordering within the stack using the `update_stack_body` function.

- `ListValue`s (a block that creates a list from items):

  - Find the match operation.

  - Handle changes and reordering within the list items using the `update_items` function.

- `WhileStatement`s:

  - Find the match operation.

  - If the condition for the while statement has changed other than through an update (meaning a new condition is being moved here), use a `BlockMove` event to move the condition here to have a parent of this while block and a new input name of "Do."

  - Handle changes and reordering within the list items using the `update_items` function.

- `ConditionalStatement`s:

  - Find the match operation.

  - Update the mutation if the number of `else` or `elseif` branches have changed and the mutation hasn't already been updated. To update the mutation, use `BlockChange` and change the `mutation` element.

  - Go through all if/dos in the modified conditional statement. Follow the process described for stack bodies, finding the correct `test` and `body` using the operations dictionaries and moving them to the correct place if they are not already there.

Separately, find the correct else body if needed and move it to the correct location as well. Update the stack bodies within each `do` as well.

- `ForListStatements` (a block that allows some stack of statements to be executed once for each item in a list):
  - Find the match operation.
  - If the `LIST` input has changed through something being inserted into the socket (rather than the existing list input being updated), move the input using `BlockMove` to the correct place (set its parent ID to this for list block and set its new input name to "LIST."
  - Handle reordering within the stack using the `update_stack_body` function.
  - Handle a change to the `VAR` input (the name of the variable used for each list item) if applicable using the `BlockChange` event to change the "VAR" field.
- `ForDictionaryStatements`s (a block that allows some stack of statements to be executed once for each key/value pair in a dictionary):
  - Find the match operation.
  - If the `DICT` input has changed through something being inserted into the socket (rather than the existing list input being updated), move the input using `BlockMove` to the correct place (set its parent ID to this for list block and set its new input name to "DICT."
  - Handle reordering within the stack using the `update_stack_body` function.
  - Handle a change to the `KEY` input (the name of the variable used for the key in each pair) if applicable using the `BlockChange` event to change the "KEY" field.
  - Handle a change to the `VALUE` input (the name of the variable used for the value in each pair) if applicable using the `BlockChange` event to change the "VALUE" field.

- `ForRangeStatements` (a block that allows some stack of statements to be executed once for each index in a range):
  - Find the match operation.
  - Handle a change to the `VAR` input (the name of the variable used for each list item) if applicable using the `BlockChange` event to change the "VAR" field.
  - Handle reordering within the stack using the `update_stack_body` function.
- `DoReturnExpressions`:
  - Find the match operation.
  - If the `VALUE` input has changed through something being inserted into the socket (rather than the existing value input being updated), move the input using `BlockMove` to the correct place (set its parent ID to this for list block and set its new input name to "VALUE."
  - Handle reordering within the stack using the `update_stack_body` function.
- `ConditionalExpressions` (such as "do x if y else z"):
  - Find the match operation.
  - If the `TEST` input has changed through something being inserted into the socket (rather than the existing test input being updated), move the input using `BlockMove` to the correct place (set its parent ID to this for list block and set its new input name to "TEST."
  - If the `THENRETURN` input has changed through something being inserted into the socket (rather than the existing test input being updated), move the input using `BlockMove` to the correct place (set its parent ID to this for list block and set its new input name to "THENRETURN."

    - If the `ELSERETURN` input has changed through something being inserted into the socket (rather than the existing test input being updated), move the input using `BlockMove` to the correct place (set its parent ID to this for list block and set its new input name to "ELSERETURN."

- `LetBlockStatement`s (a block that allows some stack of statements to be executed using some declared local variables):
    - Find the match operation.
    - Update the mutation if any of the local variable names have changed and the mutation has not already been changed.
    - Follow a similar procedure to the `update_items` function to make sure all of the local variable values are in their correct places.
    - Handle reordering within the stack using the `update_stack_body` function.

- `StatementProcedureDecl`s:
    - Find the match operation.
    - Update the mutation if any of the parameter names have changed and the mutation has not already been changed.
    - Handle reordering within the stack using the `update_stack_body` function.
    - Change the procedure name if applicable using the `BlockChange` event on the `NAME` field.

- `MethodCallStatement`s:
    - Find the match operation.
    - Look for the corresponding original method in the dictionary of procedure declarations created, in order to have access to what the original parameter names were. Also look for the modified method's parameter names. The reason this step

is necessary is because `MethodCallStatement` nodes only keep track of a list of input arguments, without any list or mapping to the parameter names other than the one that is implicit based on ordering within the list.

  - Update the mutation if any of the parameter names have changed and the mutation has not already been changed.
  - Follow a similar procedure to the `update_items` function to make sure all of the input values are in their correct places.
  - Handle reordering within the stack using the `update_stack_body` function.
  - Change the procedure name if applicable using the `BlockChange` event on the `PROCNAME` field.

- `ColorValues`:
  - Convert the value in the format "0xFF..." to the format that the blocks use, which is "#.." (with all lowercase and omitting the alpha FF).
  - Issue a `BlockChange` event to change the `COLOR` field.

- `NumberValues`:
  - Check if the node has an `aiatools_ref`. If so, this is a number block, so issue a `BlockChange` event on the `NUM` field. Otherwise, this is a number value as part of a property definition in a designer assignment, so call `__update_parent__` to update the designer assignment instead.

### Designer Insertions

Designer insertions involve inserting either `ComponentDecl`s or `DesignerAssignment`s.

For inserting a `ComponentDecl`, a `ComponentAdd` event is created with the correct name, parent ID, and insertion index. The parent is found by looking in the component dictionary for a component with the same name as the `ComponentDecl`'s value's parent's `field_name`.

For inserting a `DesignerAssignment`, a `ComponentProperty` event is created to change the specified property from its default value to the specified new value.

### Block Insertions

Block insertions involve issuing `BlockCreate` events and then calling update on the parent node to process associated changes.

For most block insertions, the blocks can be created directly from XML generated from calling the Project Writer's `convert_to_string` function on the node being inserted. This can be done with multiple blocks at once, such as if an entire bound component event with several blocks in the stack are being created.

For more complicated types like `ConditionalBranch`es where the node type being created does not directly translate into the type of block needed, additional logic is needed to create the correct XML (or avoid creating blocks if we're inserting a `ast.PassStatement` or `ast.NoneValue`), but examples of this are shown later.

The following subsections contain examples of edits.

### Example: Adding list items

Suppose the user has this list:

Figure 3.7: A code sample from an app involving a list.

which is represented as

```
initialize names = ['James', 'Emily']
```

in Aptly code. Suppose the user requests, "Add 'Ashley' as the second item in the list" to achieve this result:

```
initialize names = ['James', 'Ashley', 'Emily']
```

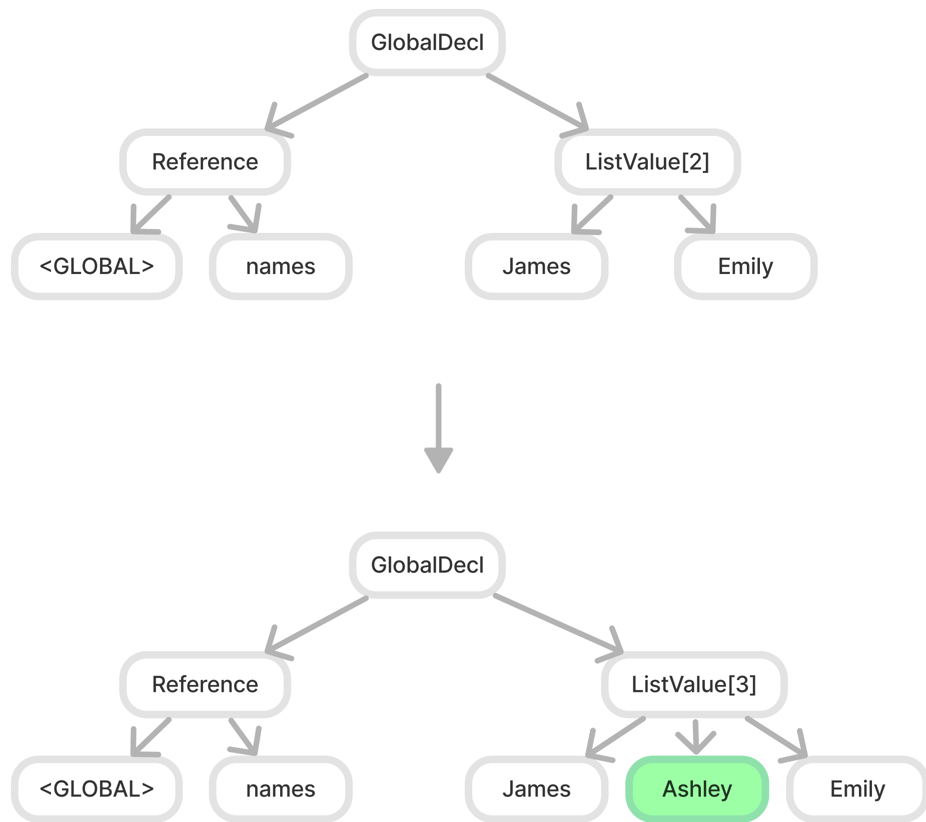


Figure 3.8: Adding an element to the middle of the list, shown at the AST level.

The change at the AST level is shown in figure 3.8.

The edit operations returned from ZSS are as follows:

```
<Operation Match> <GLOBAL> <GLOBAL> (Identifier)
<Operation Match> names names (Identifier)
<Operation Match> global names global names (Reference)
<Operation Match> James James (StringValue)
<Operation Insert> None Ashley (StringValue)
<Operation Match> Emily Emily (StringValue)
<Operation Match> ListValue[...] ListValue[...] (ListValue)
<Operation Match> global names = <aptly.ast.ListValue object at 0
   x10f10b9d0> global names = <aptly.ast.ListValue object at 0
   x10c1116f0> (GlobalDecl)
```

The steps to make the transformation are:

1. Create the ‘Ashley’ text block
2. Update the mutation of the ‘make a list’ block to have 3 sockets instead of 2
3. Move the ‘Ashley’ text box to the second socket and set its parent to the id of the ‘make a list’ block and its input name to `ITEM1`.
4. Move the ‘Emily’ text box down to the third socket and set its input name to `ITEM2`.

The edit event sequence produced is as follows:

```
{"type": "create", "blockId": "id_of_Ashley_text", "ids": ["
   id_of_Ashley_text"], "xml": "<block id=\"id_of_Ashley_text\" type
   =\"text\"><field name=\"TEXT\">Ashley</field></block>"},
{"type": "change", "blockId": "id_of_list", "element": "mutation", "
   newValue": "<mutation items=\"3\"></mutation>", "oldValue": "<
   mutation items=\"2\"></mutation>"},
```

```
{"type": "move", "blockId": "id_of_Ashley_text", "newParentId": "
    id_of_list", "newInputName": "ADD1"},
{"type": "move", "blockId": "id_of_Emily_text , "newParentId": "
    id_of_list", "newInputName": "ADD2"}
```

The resulting list is:

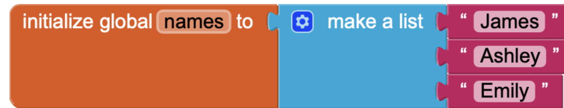


Figure 3.9: The same code sample from the previous figure with 'Ashley' inserted into the middle of the list.

### Example: Updating a conditional statement

Conditional statements (if/else if/else control blocks) require complex logic in order to update them partially due to their AST representation.

Take this figure, for example:

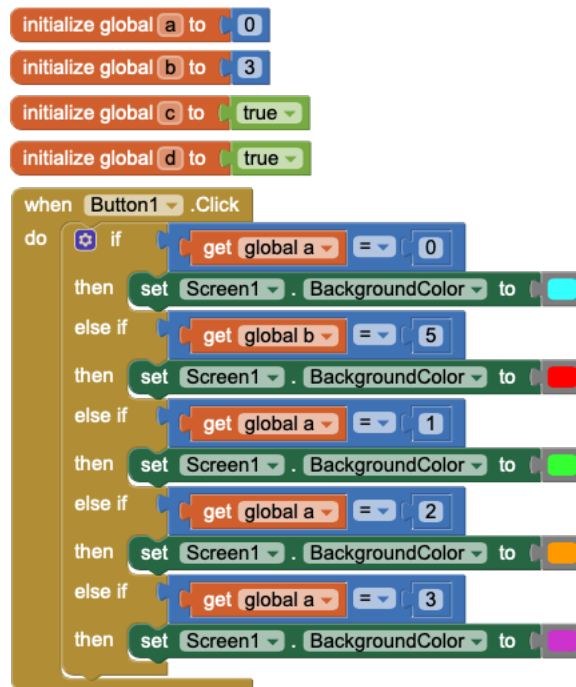


Figure 3.10: A code sample from an app involving a conditional statement.

In Aptly code, this would be represented as:

```
when Button1.Click():
  if global a == 0:
    set Screen1.BackgroundColor = Color(0xFF33FFFF)
  elif global b == 5:
    set Screen1.BackgroundColor = Color(0xFFFF0000)
  elif global a == 1:
    set Screen1.BackgroundColor = Color(0xFF33FF33)
  elif global a == 2:
    set Screen1.BackgroundColor = Color(0xFFFF9900)
  elif global a == 3:
    set Screen1.BackgroundColor = Color(0xFFCC33CC)
```

and the AST representation of the conditional section would be the following (some nodes omitted for brevity):

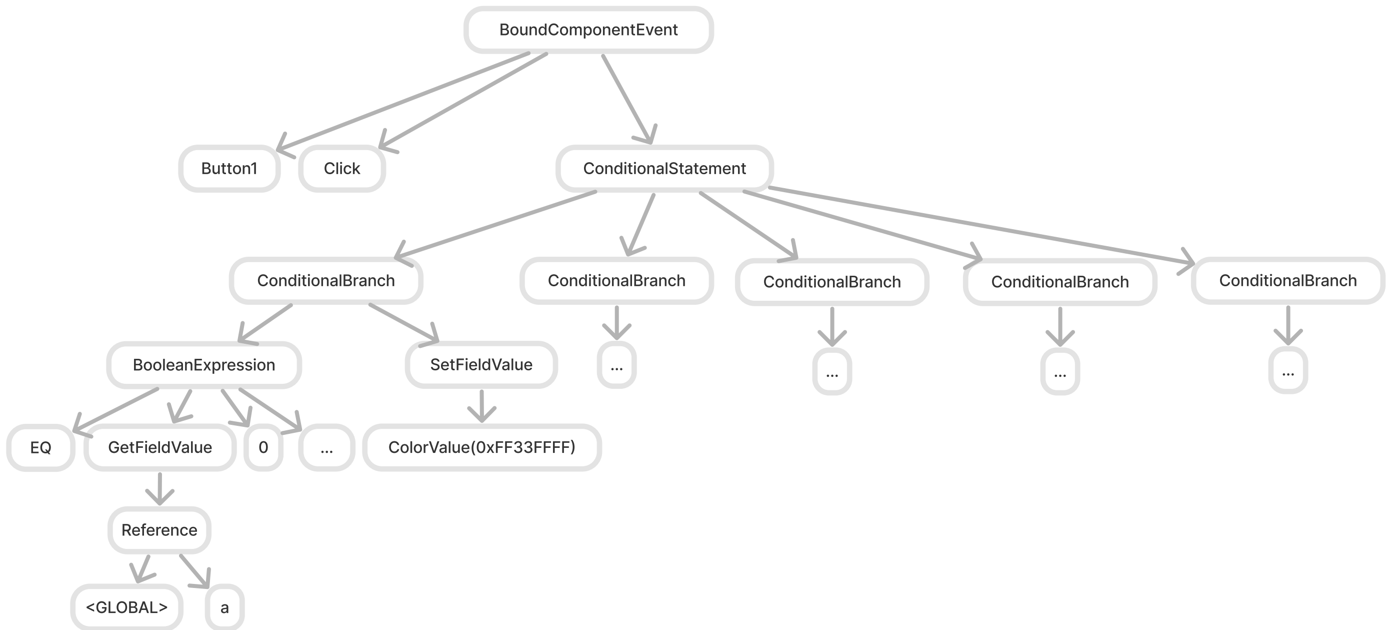


Figure 3.11: The AST representation of a bound component event containing a conditional statement.

As we can see, the "test" (the condition) and "body" (the statements executed if that

condition is true) for each else/if are grouped together in a ConditionalBranch, which is an AST node. The "test" and "body" are themselves AST nodes as well.

Suppose the user now issues a command such as "Actually, the screen should change to light blue if c is equal to d. Check if c equals d first. If a is 0, then the button needs to be disabled instead," perhaps with the expectation that the corresponding code section will turn into the following:

initialize global a to 0
initialize global b to 3
initialize global c to true
initialize global d to true
when Button1 .Click
do if get global c = get global d
then set Screen1 . BackgroundColor to
else if get global a = 0
then set Button1 . Enabled to false
else if get global b = 5
then set Screen1 . BackgroundColor to
else if get global a = 1
then set Screen1 . BackgroundColor to
else if get global a = 2
then set Screen1 . BackgroundColor to
else if get global a = 3
then set Screen1 . BackgroundColor to

Figure 3.12: The updated code sample from an app involving a conditional statement.

During ZSS, the algorithm will recognize that it needs to insert the `if global c == global d` (the 'test') and the `set Button1.Enabled = False` (the 'body') and that this also involved adding another ConditionalBranch.

It may also attempt to leave some of the setters alone and update the color value instead rather than issuing move events, or leave some of the getters in place and update the variable name instead as well. Here are the operations returned (omitting most of the `MATCH` operations for brevity).

```
<Operation Update> a c (Identifier)
<Operation Update> global a global c (Reference)
<Operation Match> GetFieldValue(Reference(Identifier('a')is_global=
   True)) GetFieldValue(Reference(Identifier('c')is_global=True)) (
   GetFieldValue)
<Operation Insert> None GetFieldValue(Reference(Identifier('d')
   is_global=True))
<Operation Remove> NumberValue('0') None
<Operation Insert> None EQ (Operator)
<Operation Insert> None <GLOBAL> (Identifier)
<Operation Insert> None a (Identifier)
<Operation Insert> None global a (Reference)
<Operation Insert> None GetFieldValue(Reference(Identifier('a')
   is_global=True))
<Operation Insert> None NumberValue('0') (<class 'aptly.ast.
   NumberValue'>)
<Operation Insert> None <aptly.ast.BooleanExpression object at 0
   x10c821630>
<Operation Insert> None Button1 (Identifier)
<Operation Insert> None Enabled (Identifier)
<Operation Insert> None Button1.Enabled (ComponentFieldName)
<Operation Insert> None BooleanValue(False) (BooleanValue)
<Operation Insert> None Button1.Enabled = BooleanValue(False) (
   SetFieldValue)
<Operation Insert> None ConditionalBranch(<aptly.ast.
   BooleanExpression object at 0x10c821630>, [<aptly.ast.
   SetFieldValue object at 0x10c821840>])
```

Figure 3.13 shows the changes at the AST level.

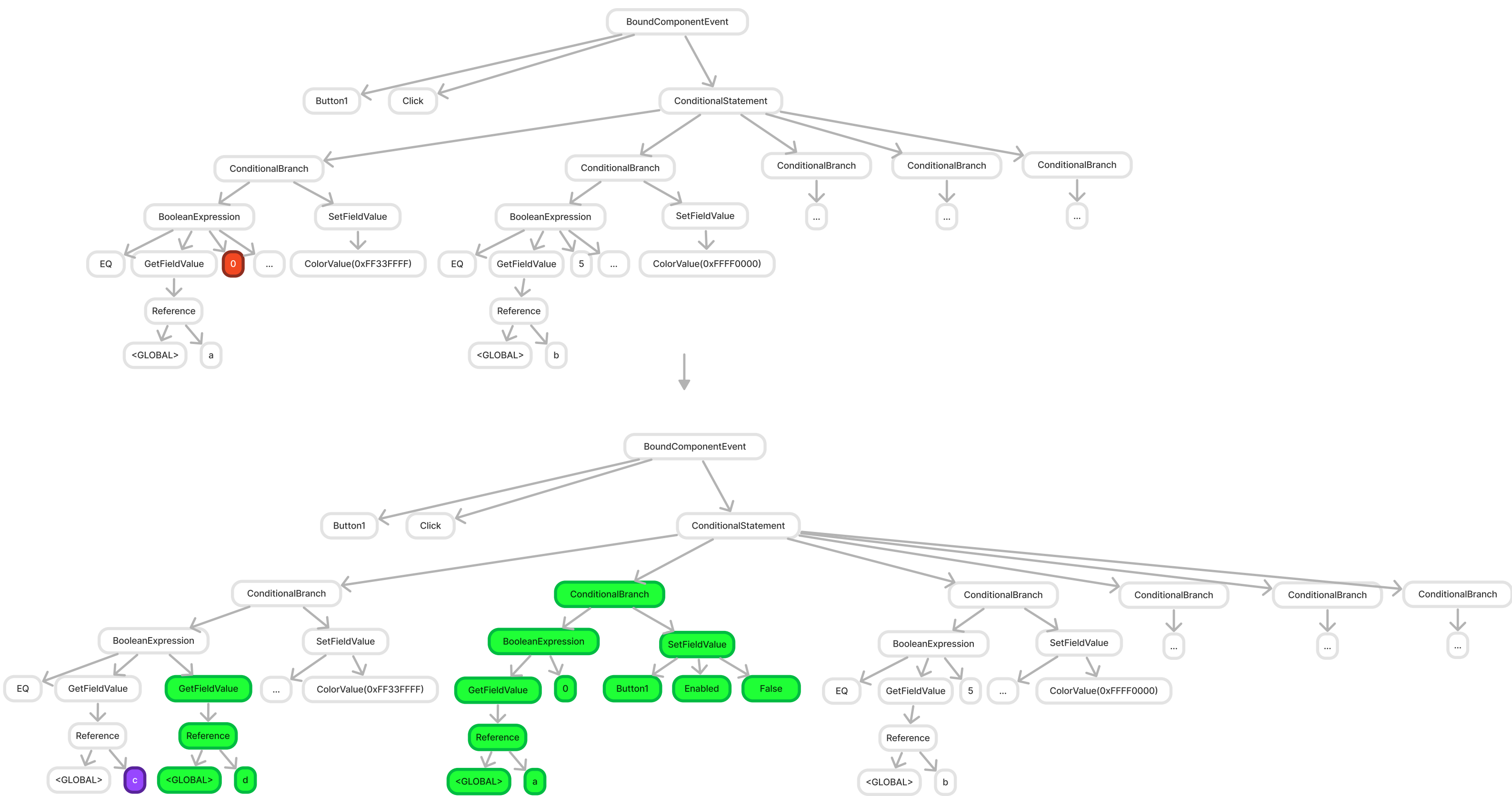


Figure 3.13: The AST node changes for the bound component event involving a conditional statement. Insertions are shown in green, removals are shown in red, and updates are shown in purple.

However, in MIT App Inventor, the way the blocks are set up, the changes that are needed are:

1. Update the mutation of the control block holding the if/else if/thens. The mutation in XML looks like

   ```
   <mutation xmlns=\"http://www.w3.org/1999/xhtml\" elseif=\"5\"></mutation>
   ```

   and it needs to be changed to

   ```
   <mutation xmlns=\"http://www.w3.org/1999/xhtml\" elseif=\"6\"></mutation>
   ```

2. Insert a boolean expression where the operator is `=` and the two operands are getters for the global variables c and d, respectively.

3. Insert a setter for `Button1.Enabled` connected to a `false` value block

4. Move the boolean expression to the right place, meaning it is in the first if's socket, its parent ID is the ID of the conditional expression block, and its input name is `IF0`.

5. Move the setter for `Button1.Enabled` to the right place, meaning it is connected to the first then, its parent ID is the ID of the conditional expression block, and its input name is `DO1`.

6. Move all of the other boolean expressions down by one socket and increment their input name by 1 (such as from `IF2` to `IF3`).

7. Move all of the other setters down to the next then and increment their input name (such as from `DO2` to `DO3`).

Here is the full sequence of edit events emitted:


```
{"type": "change", "blockId": "id1", "element": "field", "name": "VAR
  ", "newValue": "global c", "oldValue": "global a"},
```

```
{"type": "create", "blockId": "id2", "ids": ["id2", "id3", "id4"], "
   xml": "<block id=\"id2\" type=\"logic_compare\"><field name=\"OP
   \">EQ</field><value name=\"A\"><block id=\"id3\" type=\"
   lexical_variable_get\"><field name=\"VAR\">global a</field></block
   ></value><value name=\"B\"><block id=\"id4\" type=\"math_number
   \"><field name=\"NUM\">0</field></block></value></block>"},
{"type": "create", "blockId": "id5", "ids": ["id5", "id6"], "xml": "<
   block id=\"id5\" type=\"component_set_get\"><mutation
   component_type=\"Button\" is_generic=\"false\" set_or_get=\"set\"
   property_name=\"Enabled\" instance_name=\"Button1\" /><field name
   =\"COMPONENT_SELECTOR\">Button1</field><field name=\"PROP\">
   Enabled</field><value name=\"VALUE\"><block id=\"id6\" type=\"
   logic_boolean\"><field name=\"BOOL\">FALSE</field></block></value
   ></block>"},
{"type": "change", "blockId": "id_of_conditional_stmt", "element": "
   mutation", "newValue": "<mutation elseif=\"5\" else=\"0\"></
   mutation>", "oldValue": {"elseif": "4"}},
{"type": "move", "blockId": "id_of_c_d_comparison", "newParentId": "
   id_of_conditional_stmt", "newInputName": "IF0"},
{"type": "move", "blockId": "id2", "newParentId": "
   id_of_conditional_stmt", "newInputName": "IF1"},
{"type": "move", "blockId": "id5", "newParentId": "
   id_of_conditional_stmt", "newInputName": "DO1"},
{"type": "move", "blockId": "id_of_b_5_comparison", "newParentId": "
   id_of_conditional_stmt", "newInputName": "IF2"},
{"type": "move", "blockId": "id_of_set_color_red", "newParentId": "
   id_of_conditional_stmt", "newInputName": "DO2"},
{"type": "move", "blockId": "id_of_a_1_comparison", "newParentId": "
   id_of_conditional_stmt", "newInputName": "IF3"},
```

```
{"type": "move", "blockId": "id_of_set_color_green", "newParentId": "
    id_of_conditional_stmt", "newInputName": "DO3"},
{"type": "move", "blockId": "id_of_a_2_comparison", "newParentId": "
    id_of_conditional_stmt", "newInputName": "IF4"},
{"type": "move", "blockId": "id_of_set_color_orange , "newParentId": "
    id_of_conditional_stmt", "newInputName": "DO4"},
{"type": "move", "blockId": "id_of_a_3_comparison", "newParentId": "
    id_of_conditional_stmt", "newInputName": "IF5"},
{"type": "move", "blockId": "id_of_set_color_purple", "newParentId":
    "id_of_conditional_stmt", "newInputName": "DO5"},
{"type": "create", "blockId": "id7", "ids": ["id7"], "xml": "<block
    id=\"id7\" type=\"lexical_variable_get\"><field name=\"VAR\">
    global d</field></block>"}
```

#### Example: Updating the argument names for a user-defined procedure

Suppose the user starts out with this code snippet:

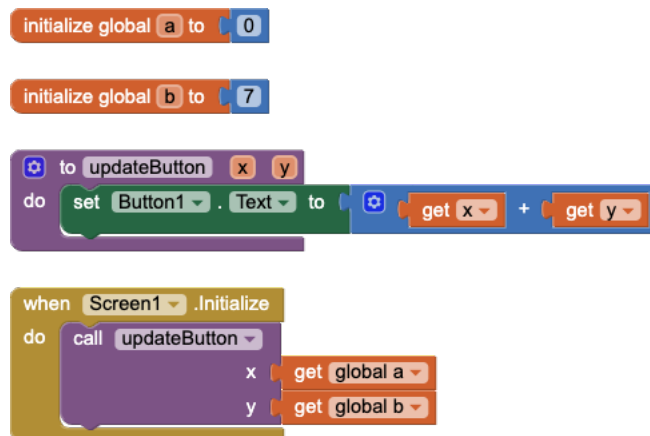


Figure 3.14: Code blocks for an app that uses a procedure to update a button's text

which translates to this Aptly code:

```
Screen1 = Screen(Title = 'Screen1')
Button1 = Button(Screen1, Text = 'Text for Button1')
```

```
initialize a = 0
initialize b = 7

to updateButton(x, y):
  set Button1.Text = x + y

when Screen1.Initialize():
  call updateButton(global a, global b)
```

If the user issues the command "Add another parameter called 'new_arg' and make it the first element. Pass in 5 for this argument when the procedure is called" then the resulting Aptly code would likely look like this:

```
Screen1 = Screen(Title = 'Screen1')
Button1 = Button(Screen1, Text = 'Text for Button1')

initialize a = 0
initialize b = 7

to updateButton(new_arg, x, y):
  set Button1.Text = x + y

when Screen1.Initialize():
  call updateButton(5, global a, global b)
```

At the AST level, this change looks like the following:

And the sequence of operations returned from ZSS would be all `Match` operations except for these two:

```
<Operation Insert> None new_arg (Identifier)
<Operation Insert> None NumberValue('5')
```

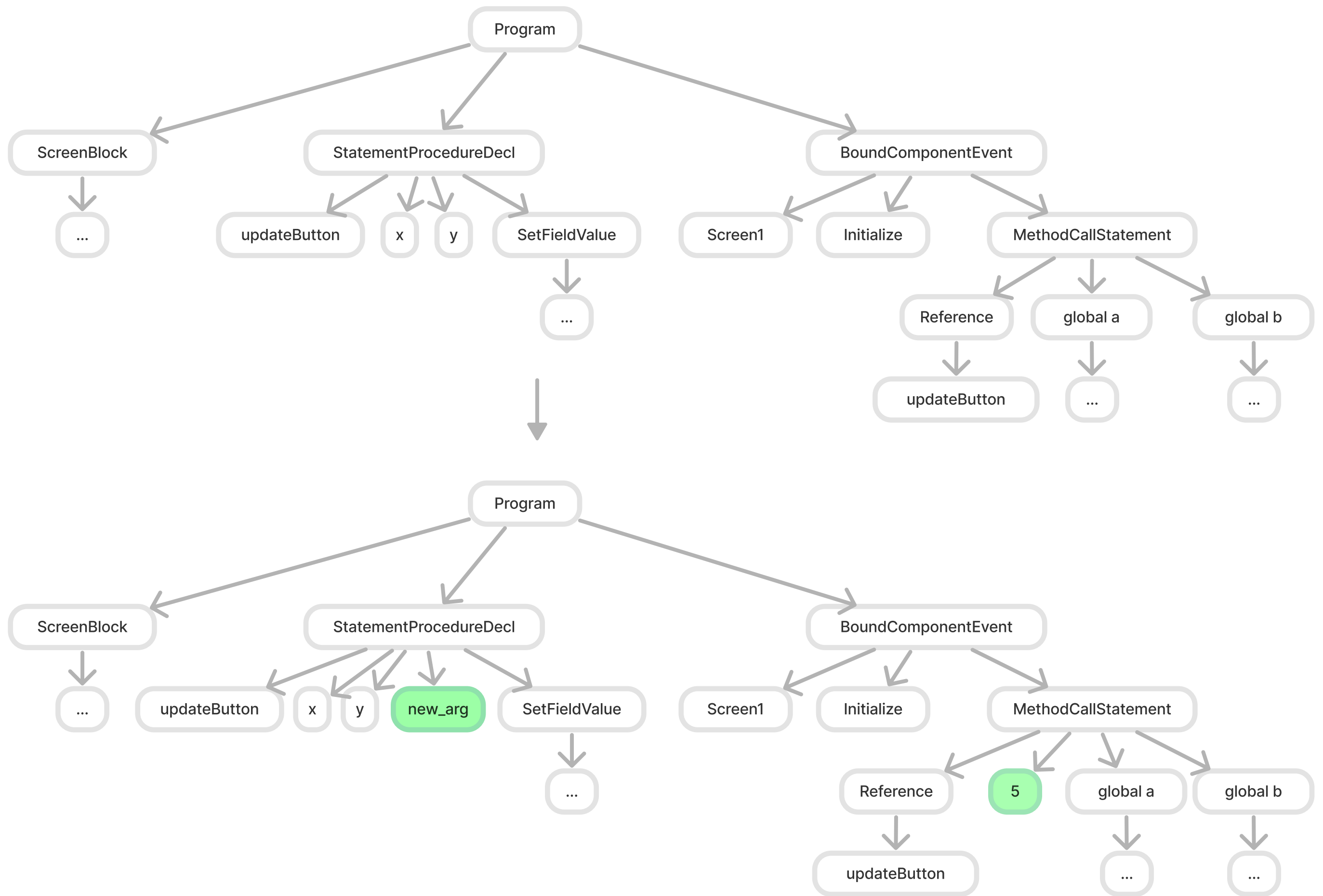


Figure 3.15: The AST representations of the original and modified app, with the insertions in green.

The process for making this change is more complicated than just two insertions:

1. Create a number block with value 5.
2. Insert the `Identifier` for `new_arg` and then call update on the parent, which is the `StatementProcedureDecl`. This will result in the following changes:
   (a) Update the `StatementProcedureDecl`'s mutation to include the new argument.
   (b) Update the `MethodCallStatement`'s mutation to include the new argument.
   (c) Set `5`'s input name to `ARG0` and set its parent pointer to the ID of the `MethodCallStatement`.
   (d) Change `global a`'s input name from `ARG0` to `ARG1`.
   (e) Change `global b`'s input name from `ARG1` to `ARG2`.

```
{"type": "change", "blockId": "stmt_proc_decl_id", "element": "
   mutation", "newValue": "<mutation><arg name=new_arg></arg><arg
   name=x></arg><arg name=y></arg></mutation>", "oldValue": <mutation
   ><arg name=x></arg><arg name=y></arg></mutation>},
{"type": "create", "blockId": "new_arg_id", "ids": ["new_arg_id"], "
   xml": "<block id=\"new_arg_id\" type=\"math_number\"><field name
   =\"NUM\">5</field></block>"},
{"type": "change", "blockId": "stmt_proc_call_id", "element": "
   mutation", "newValue": "<mutation name=updateButton><arg name=
   new_arg></arg><arg name=x></arg><arg name=y></arg></mutation>", "
   oldValue": "<mutation name=updateButton><arg name=x></arg><arg
   name=y></arg></mutation>"},
{"type": "move", "blockId": "new_arg_id", "newParentId": "
   smt_proc_call_id, "newInputName": "ARG0"},
{"type": "move", "blockId": "x_arg_id", "newParentId": "
   smt_proc_call_id", "newInputName": "ARG1"},
```

```
{"type": "move", "blockId": "y_arg_id", "newParentId": "
   smt_proc_call_id", "newInputName": "ARG2"}
```

**Example: Deleting a component with blocks**

In MIT App Inventor, deleting a component deletes all of its children as well as all code blocks associated with that button. For example, given this initial app:

```
Screen1 = Screen(Title = 'Screen1')
Button1 = Button(Screen1, Text = 'Text for Button1')
Button2 = Button(Screen2, Text = 'Text for Button2')

when Button1.Click():
    set Button1.BackgroundColor = ColorValue(0xFF33FF33)

when Button2.Click():
    set Button1.BackgroundColor = ColorValue(0xFF3333FF)
```

if the user requests to delete Button1, the resulting Aptly code will be as follows:

```
Screen1 = Screen(Title = 'Screen1')
Button2 = Button(Screen1, Text = 'Text for Button2')

when Button2.Click():
    pass
```

Figure 3.16 shows this change at the AST level:

The operations returned from ZSS are:

```
<Operation Remove> Button1 None (Identifier)
<Operation Remove> Button1 None (Reference)
<Operation Remove> Button None (Identifier)
<Operation Remove> Screen1 None (Identifier)
```

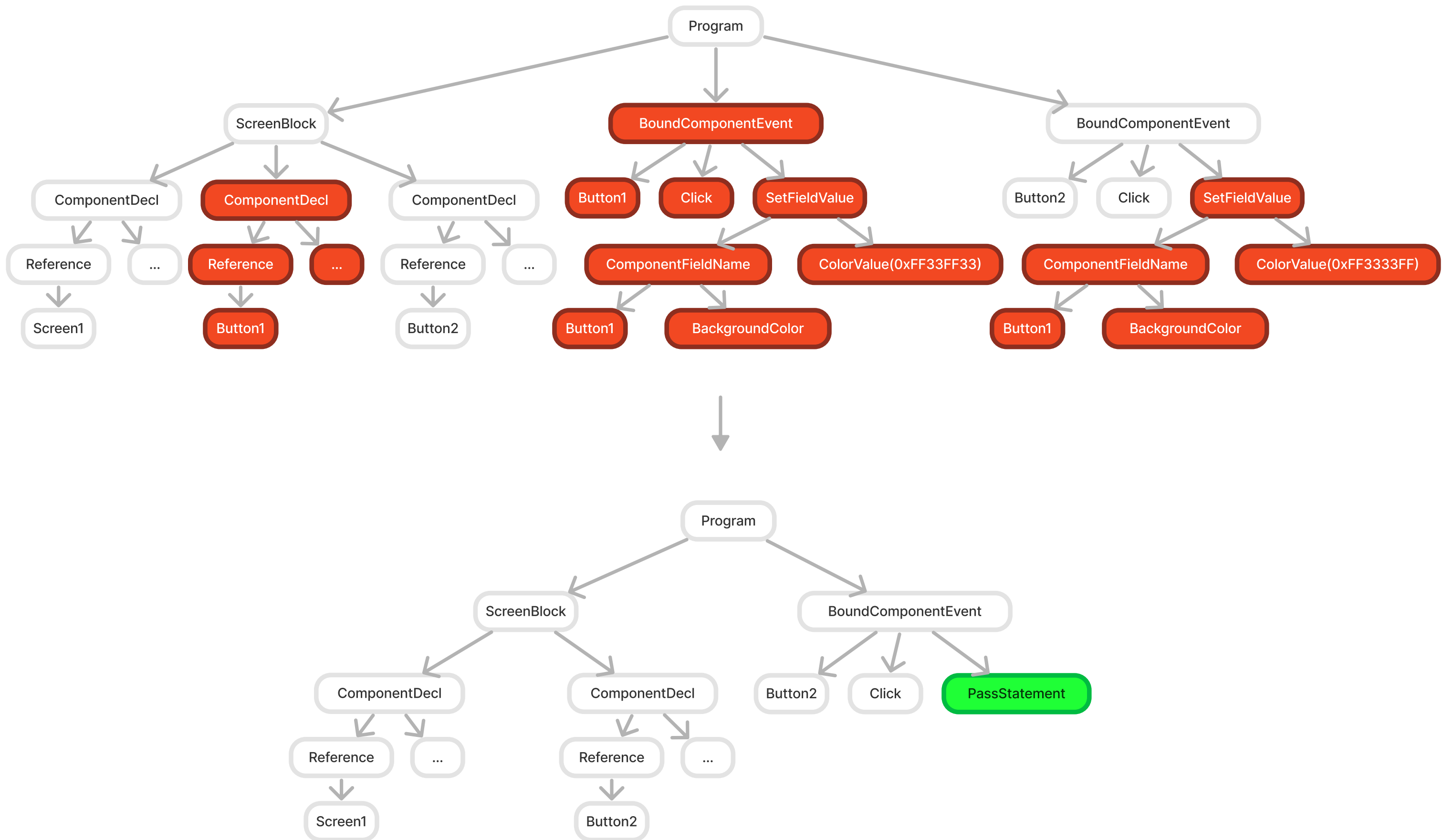


Figure 3.16: The AST representations of the original and modified app, with the deletions in red and the insertion in green.

```
<Operation Remove> Screen1 None (Reference)
<Operation Remove> Text None (Identifier)
<Operation Remove> Text None (Reference)
<Operation Remove> Text for Button1 None (StringValue)
<Operation Remove> Text = Text for Button1 None (DesignerAssignment)
<Operation Remove> ComponentTypeInvocation(Identifier('Button'),
   Reference(Identifier('Screen1')), {Identifier('Text'): StringValue
   ('Text for Button1')}) None
<Operation Remove> Button1 = ComponentTypeInvocation(Identifier('
   Button'), Reference(Identifier('Screen1')), {Identifier('Text'):
   StringValue('Text for Button1')}) None (ComponentDecl)
<Operation Remove> Button1 None (Identifier)
<Operation Remove> Click None (Identifier)
<Operation Remove> Button1 None (Identifier)
<Operation Remove> BackgroundColor None (Identifier)
<Operation Remove> Button1.BackgroundColor None (<class 'aptly.ast.
   ComponentFieldName'>)
<Operation Remove> ColorValue('0xFF33FF33') None
<Operation Remove> Button1.BackgroundColor = ColorValue('0xFF33FF33')
    None (SetFieldValue)
<Operation Remove> BoundComponentEvent(Identifier('Button1'),
   Identifier('Click')) None
<Operation Remove> Button1.BackgroundColor None (ComponentFieldName)
<Operation Insert> None Pass (PassStatement)
<Operation Remove> ColorValue('0xFF3333FF') None
<Operation Remove> Button1.BackgroundColor = ColorValue('0xFF3333FF')
    None (SetFieldValue)
```

In this case, however, the edit events needed are actually much simpler than the sequence

of operations returned.

There is no block corresponding to `ast.PassStatement` in MIT App Inventor; this AST node is only created for allowing the parsing of Aptly code where the corresponding MIT App Inventor app would have an empty socket or empty body.

Additionally, in MIT App Inventor, deleting a component deletes all associated blocks, so it is only necessary to issue a single component deletion event for Button1 which will delete all blocks corresponding to the button as well.

### 3.2.10 Applying edit events to the project

In order to implement the edits automatically and change the UI and/or blocks of the app the user is currently working on, the Real-Time Collaboration (RTC) system previously developed for App Inventor [13] is used.

#### The Real-Time Collaboration (RTC) System

MIT App Inventor has a Real-Time Collaboration (RTC) system [13] which allows multiple users to collaborate on the same MIT App Inventor project at the same time. The RTC system works using WebSockets, and users join the channel corresponding to the current project.

Events are received by the RTC Server and processed by a `CollaborationManager`.

These events translate into events like those issued by one user working in MIT App Inventor, such as the creation of code blocks and the removals of components.

#### Utilizing the RTC System

Once ZSS returns a sequence of edit events (inserting, deleting, or updating nodes of the AST), the Aptly server can act as another user collaborating on the project and emit those events to App Inventor through the RTC channel; Aptly joins the project channel as “aptly@appinventor.mit.edu.”

The `CollaborationManager` examines incoming events, and for events where the user is "aptly@appinventor.mit.edu," it marks the `isAI` field as true.

The events are then applied in the same way a user-issued event would be applied.

### Aptly Project Edit Undo Wizard

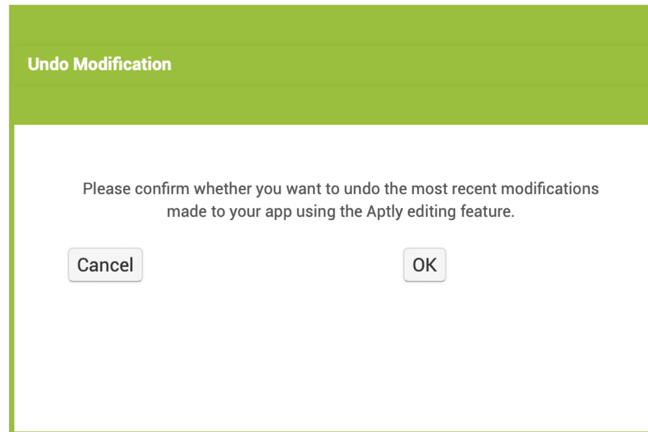


Figure 3.17: The Aptly Project Edit Undo Wizard

The blue undo button is disabled initially. When the user makes an edit via Aptly, the button becomes enabled. If the user then manually makes edits after Aptly's edit, the button becomes disabled again, but if the user undoes those changes using the undo option in the Blocks editor, then the blue undo button becomes enabled again.

This functionality is implemented by checking, upon each event issued by either the user or Aptly, if the top of the undo stack is an event with the `isAI` field set.

The App Inventor server keeps track of two separate undo stacks, one for the Designer and one for the Blocks editor. At the beginning of an Aptly-issued edit, a separate undo stack is created to hold all events generated during the edit, and then the resulting stack is concatenated onto the existing designer and blocks undo stacks.

To determine whether the undo button should be enabled or disabled, upon each received event, the top events of both the designer and blocks undo stacks are examined. Here are the checks made:

1. If both the designer and blocks' undo stacks are empty, the button is disabled.

2. If the designer undo stack is empty and the blocks' undo stack is not empty, then the button is enabled if and only if the top block event has a true `isAI` field.

3. If the blocks undo stack is empty and the designer undo stack is not empty, then the button is enabled if and only if the top designer event has a true `isAI` field.

4. If both the designer and blocks undo stacks are nonempty, then the button is enabled if and only if the most recent event out of the top event on each stack (the event with the most recent timestamp) has a true `isAI` field.

When the user clicks the blue undo button, the undo function for both the Blockly workspace (implemented by the Blockly developers with additional scaffolding by the MIT App Inventor team) and the Designer (implemented partially by myself, with some aspects implemented by other team members prior to my involvement) are called, which automatically update the blocks and designer undo stacks, respectively.

# Chapter 4

# Study Design

A user study was conducted to observe how participants would utilize the tool to complete three given tasks, namely replicating a specific app, designing an app to solve a particular problem, and creating an app of their choice.

In addition to gathering general observations, one specific goal of the study was to observe participants' workflows and the extent to which participants would utilize Aptly Editing vs. manually editing the apps themselves.

## 4.1 Participants

Participants were high school students with varying levels of programming experience, selected with the goal of seeing how individuals with no programming experience vs. extensive programming experience interact with the tool.

The demographics of the participants were as follows:

- **Age:** 17 (5), 18 (5)
- **General programming experience:** None (3), $<$ 1 year (2), 3-4 years (3), 5+ years (2)
- **Familiarity with block coding:** None (3), Beginner (2), Familiar (5)

## 4.2 Survey Questions

In addition to demographic questions, participants were asked the following questions:

- How interested would you be in learning how to program?
- Given the right tools, could you see yourself making your own interesting app by yourself?
- If you could make any mobile app right now, what would it be? (don't restrict yourself on what you think is possible - just imagine the best app you want to make) give a brief description
- What would you say is the biggest factor that stops most people from learning programming?
- How strongly would you agree with this statement: "Programming is accessible for everyone."

## 4.3 Learning the basics of the tool

After completing the pre-survey, students were briefly shown MIT App Inventor and shown that they can drag and drop components to design the user interface in the Designer tab, and how they can drag and drop blocks in the Blocks tab to implement the functionality of their app. The reasoning for showing participants the basics of MIT App Inventor was to reduce confounding effects on the study caused by students being confused about aspects of the design of MIT App Inventor that weren't related to the design of Aptly.

Participants were additionally shown how to create a project from a description (Aptly App Creation) and how to click the pencil icon and enter an edit description (Aptly Editing). Participants were then given five minutes to play around with the tool and become familiar

with its basic mechanics. This time was provided with the goal of having the observations about participants' interactions with the tool be more representative of how students would use it in the steady state, rather than how someone who has no idea how to use the tool would fumble around with it for the first few minutes.

## 4.4 Task 1: Replicate a specific app

For the first task, participants were shown an app which features three buttons (red, green, and blue) and lets users draw on the screen in the selected color after the user clicks a button. For instance, if the user clicks the red button, they can draw in red, and if the user now clicks the blue button, the pen color changes to blue and they can continue drawing in blue.

Participants were asked to recreate this app using Aptly. Each participant had the opportunity to begin by describing the app to Aptly App Creation, and then participants were free to use any combination Aptly Editing and manually dragging and dropping components and blocks.

This task assessed participants' ability to achieve a specific desired end result given the tool.

## 4.5 Task 2: Create an app to solve a given problem

The second task was more open-ended, but still structured. Participants were presented with the following scenario: "In your class you have a friend who is visually impaired and is having trouble with their math class. You want to help that student by creating a calculator app that will help them with their addition and subtraction. Try using Aptly to create that app. Feel free to use the editing and also manually drag and drop the blocks."

This task assessed how participants utilized Aptly to not only create an app from a technical standpoint but also potentially participate in the creative process. With a more open-ended goal, students had the opportunity to leave creative aspects of the app (i.e. the

best way to go about making the calculator work for the visually-impaired individual) up to Aptly rather than students having to think through these design aspects themselves.

## 4.6 Task 3: Create an app of the participants' choosing

The final task was aimed at seeing what types of apps students would like to make, and whether and how the tool empowers students to make those apps.

The instructions given to participants for this task was "Make any app you want." Participants had the opportunity to use any combination of Aptly or manually dragging and dropping components and blocks.

# Chapter 5

# Results & Discussion

## 5.1 Summary of participants' initial attitudes towards programming

Prior to the study, students expressed a variety of views regarding programming.

| Response | Count |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 3 | 6 |
| 4 | 1 |
| 5 | 2 |

Table 5.1: Pre-Study Survey: Responses to "How interested (1-5) would you be in learning how to program?"

| Response | Count |
|---|---|
| Yes | 6 |
| Maybe / I don't know | 4 |

Table 5.2: Pre-Study Survey: Responses to "Given the right tools, could you see yourself making your own interesting app by yourself?"

| Response | Count |
|---|---|
| General difficulty of programming for most people | 7 |
| General lack of interest in programming | 2 |
| Lack of online resources to start learning about programming | 1 |

Table 5.3: Pre-Study Survey: Responses to "What would you say is the biggest factor that stops most people from learning programming?"

| Response | Count |
|---|---|
| Strongly agree | 2 |
| Somewhat agree | 4 |
| Neutral | 2 |
| Somewhat disagree | 1 |
| Strongly disagree | 1 |

Table 5.4: Pre-Study Survey: Sentiment of responses to "How strongly would you agree with this statement: 'Programming is accessible for everyone.' "

A reason cited for agreement with the statement "Programming is accessible for everyone" was "there are a lot of videos online that teach people and resources that people can have access to without paying money." Three participants cited the lack of universal access to technology as a reason for disagreement, specifically stating that "not everyone has wifi or a computer to use for online programming," "Kinds[sic] in some countries don't have access to computer hardware, or a computer education," and "access to technology is not universal."

When asked, "If you could make any mobile app right now, what would it be? (don't restrict yourself on what you think is possible - just imagine the best app you want to make) give a brief description," participants reported a variety of creative and ambitious ideas:

- "I would make an app that is able to find stuff that you have lost. Like it would be able to locate almost any item you had lost."
- "I'd like to make an app for artists to upload their work (drawings/paintings/etc.). Similar to Instagram or FaceBook, however it is strictly for art photos and video

uploading."

- "Probably something with food delivery– I want to make an app that makes food delivery more sustainable like with bikes or something."
- "I would make a app like Google Maps but more customizable for each user. A user could input their preferences, like if they would rather have a easy but longer (driving in straight line) vs. harder but shorter (lots of merging lanes), and the map would give them the answer. The app would also have a function to show which gas stations are nearest and best for the user (user could input if they want a near and expensive one or a far but cheap one)."
- "Pokemon Go but you take pictures of cool rocks at the side of the road"
- "an app that can read my mind and draw out what I'm picturing"
- "A fantasy game where you choose your own adventure, but it truly is open ended where you can decide if you want to be a night, a magic user, etc with endless choices– like a mini-life in a different world in a game."
- "Social media app that schedules random times to call with old friends you haven't spoken to in a while."
- "A Musical Notation App"

The variety and creativity of these app ideas demonstrates that individuals may possess creative ideas that would solve tangible problems, evidencing the value in tools that help participants realize these ideas.

## 5.2 Task 1 (Recreating an app)

Overall, participants successfully recreated the desired app. Some participants achieved a result that looked and functioned very similarly to the end goal app shown to them by

providing a short description to the initial creation process. Example successful descriptions from participants are as follows:

- "Make an app that has three different color buttons at the top of the screen that someone can click on and then draw in that color."

- "This app has three buttons, labeled 'red', 'blue' and 'green'. The button labeled 'red' is colored red, the button labeled 'green' is green, and the button labeled 'blue' is blue. These three buttons are lined up horizontally at the top of the screen. When the user clicks a button, they have the capability to draw on the screen below the buttons in the color assigned to the button. When they click another button, they lose the capability to draw in the initial color and now they have the ability to draw in the new selected color."

- "An app with 3 buttons (one red, one blue, one green), and a pen drawing feature with a blank 'drawing' space beneath it. When the mouse is dragged in the blank space, create a line where the drag is. When red is clicked, the pen color is red. When blue is clicked, the pen color is blue. When green is clicked, the pen color is green."

- "An app that has three buttons, one saying red, one saying blue, and one saying green. When you click on each of these buttons, your pen will turn into that color and you will be able to draw with that color in the white space below the three buttons."

- "This app has three buttons that are red, green and blue at the top and when you click one, it you can draw on the screen with the buttons respective color."

- "Draws with RGB."

Ultimately, Aptly App Creation was able to successfully create the desired app (or one with similar functionality) given a wide range of descriptions from participants, ranging from specific descriptions about the layout and color of the buttons to simple descriptions such as "Draws with RGB."

These results indicate that users of the tool need not be experts or phrase their descriptions in one specific way in order for GPT-4 to understand their intention. However, some participants did struggle to articulate the app in words. For example, one participant wrote "draw a line when I click a button" rather than indicating that the user should be able to draw in the specified color after the button is clicked. This subtle difference highlights a limitation of AI-powered tools such as this one; even if an AI-based tool becomes completely proficient at doing exactly what the user describes, that does not mean that the tool will be able to figure out what the user really means if the instructions issued by the user do not match the user's actual mental model. While most participants described what they wanted in a way that was understandable for Aptly, some participants were less successful in terms of describing the intended app functionality.

Some participants opted to create the app more incrementally, utilizing Aptly Editing to a greater extent. For example, one participant initially created the app with the following description: "Have three medium sized buttons on top of the screen. One button says red, the other blue, the last green." Once Aptly had created this initial app, the participant issued the following sequence of commands:

1. "The area under the buttons is a canvas where the use[sic] can draw."
2. "When the user clicks and drags, they can draw on the canvas"
3. "When the user clicks a button, their canvas drawing color changes to the respective color of the button, allowing them to draw on the canvas with the color"
4. "Align the buttons to the center of the screen"
5. "Flip everything horizontally"

Essentially, this participant began by creating the user interface, first with just the buttons and then adding the canvas. Next, the participant added functionality where the canvas

served as a drawing pad. Then, the participant implemented the functionality of the buttons, making them change the pen color when clicked. Finally, the participant aesthetically improved the app by adjusting the placement of the buttons on the screen. This methodical approach of first adding the basic components, then connecting the components together in terms of functionality, and then adding finishing touches mimics the type of incremental development style seen in many developers. In the end, this participant's app matched the desired end result. Overall, this participant's success with Aptly Editing highlights the value in providing a tool by which students can incrementally develop and improve an app.

Success with incremental development was also achieved by the participant (without prior programming experience) who wrote "draw a line when I click a button" and did not express awareness that the resulting behavior was incorrect. This participant's next command after "draw a line when I click a button" was "Draw a line with the color of the button when you click a button" as their next command, indicating an awareness that being more specific would help Aptly produce the app they desired, as well as an ability to make progress towards the end goal utilizing small improvements.

The differing extents to which, and manners in which, participants interacted with the tool indicate that providing students with a tool such as Aptly may not necessarily result in major changes to students' workflow. As an example, several of the students who created the app incrementally with one command at a time had a background in programming, and were more likely used to a development workflow of testing incrementally rather than trying to write an entire working program in one try before testing anything.

## 5.3 Task 2 (Solving a given problem)

With this more open-ended task, participants' initial descriptions ranged from a single sentence to long paragraphs about the exact layout of buttons on the screen.

Here are the app descriptions provided by the participants:

- "Make an app that can perform addition and subtraction and receives the numbers to be added or subtracted by recording someone's voice. Make it so the app reads the sum or difference out loud"

- "This app takes in vocal signals and transcribes it into calculator text, then reads the text aloud. It has the ability to audibly recognize numbers, addition, and subtraction commands. If a person states a number, it will transcribe that number. If the person states 'plus', it will transcribe '+'. If the person states 'minus', it will transcribe '-'. If the person states 'end', it will stop transcribing. Once the person states 'end', it will perform a calculating function to calculate the resul[sic] of the transcribed expression. Then, it will audibly read the full expression and solution (ie. 'four plus six equals ten', where 'ten' is the solution to the expression 'four plus six')

- "An app that can calculate basic math (addition and subtraction of numbers 0-10000, positive and negative) and that inputs information audibly and outputs information (otherwise known as the "answer", audibly). This app with[sic] have a blank screen.

- "This app will look like a calculator, but its only functions will be addition and subtraction, IN the center, there will be nine buttons, ordered in rows of three. These buttons will have the numbers 1-9 on them. Below these rows will be another button of the same size with a 0 on it. When you click any of these buttons, the number on the button will appear into the white space above all the buttons. Additionally, there will be two buttons on the left of the three rows of buttons with "-" and "+" on them. When you click these buttons, the "-" or "+" will appear in the space above like the numbers. Lastly, on the right of the three rows, there will be one other button with a "=" on it. When you press this button, the equation that is in the space above the rows will be calculated"

- "Create an app that is a calculator"

- "Create a calculator for visually impaired person"
- "Make a calculator with buttons and a display that can do addition and subtraction"
- "Make an app with a textbox near the top of the screen. Underneath, create 9 buttons. The first button has a text of '1.' The second one has a text of '2.' The third has '3.' These numbers continue, with button 4 to button 9 will have a text of '4-9".
- "There will be an output at the top of the screen that will display different numbers and symbols inputted. Underneath there are the numbers 0-9 arranged in a grid that also speak what number it is when pressed. There will also be an addition symbol, a subtraction symbol, a multiplication symbol, a division symbol, and a period symbol that all speak what symbol they are when pressed. These numbers and symbols can be combined to do different types of math, and each of these symbols and numbers when pressed will display in the outputbox at the top. There will also be an equal sign that also speaks what it is, and after being pressed will change the output box to the result of the math being done, and that result will be read out loud."

As seen here, participants' descriptions ranged from "Make an app that is a calculator" to long paragraphs about the exact layout of buttons on the screen.

Like in the last task, some participants opted to create the user interface using Aptly App Creation and then continue to develop the user interface as well as implement the app functionality using Aptly Editing, while other participants described the entire user interface and app functionality upfront during the Aptly App Creation process and then issued commands to Aptly Editing aimed at adjusting their app to better match their vision. For example, one participant chose to begin with "Make a calculator with buttons and a display that can do addition and subtraction" and then issue commands to adjust the user interface and make it more usable for a visually-impaired user.

Participants had a variety of ideas for making the calculator usable for a visually-impaired user. One participant's idea involved having the user tap the screen a certain number of times

to indicate a certain number or operation, and having the phone vibrate a certain number of times to express the sum or difference. Other participants opted for the voice recognition and text-to-speech capabilities for soliciting input from the user and providing the output to the user, respectively. Yet other students used the text-to-speech capability for providing the result to the user, but required the user to type in a text box or press buttons on the screen in order to input their addition or subtraction question. These participants had specific visions for how to best support a visually-impaired user, and described this vision to Aptly.

In contrast, the participant whose description was "Create a calculator for visually impaired person" implicitly asked Aptly to collaborate in not only the technical implementation of the app but also the creative process. This participant implicitly asked Aptly to formulate a plan for making a calculator that was suitable for a visually-impaired user; this participant essentially described the problem at hand to Aptly and asked Aptly to come up with the solution, rather than coming up with an idea for the solution themselves and asking Aptly to implement it.

The variety of ways and extents to which participants utilized Aptly to help them create this task demonstrates that tools such as Aptly can be used more or less heavily by different individuals, and that when provided with such a tool, some users will completely outsource both the creative process and the implementation process to the AI while others still prefer having control over the app's features and visual appearance.

## 5.4 Task 3 (Creating any app)

For the open-ended task asking participants to make any app they wanted, some students struggled to come up with an app idea, while others jumped in right away.

Example apps created by the participants are as follows:

1. An app to display an interactive form of a meme that rose to popularity in 2020.

2. An app that lets users draw pixel art, where the user clicks the screen and it draws a

square in one of 10 colors depending on which color button was pressed most recently.

3. An app to upload images of artwork with captions
4. An app to provide the fastest route to travel to restaurants via bicycle
5. An app to try different outfit combinations on a model (two participants separately came up with this idea)
6. An app to identify the geographically-closest gas station to the user
7. An app to control a character and interact with characters on a screen
8. An app for collecting Pokemons
9. (The final participant reported that they could not come up with an app idea)

Participants' ideas ranged from meme apps to serious apps. Many of the apps the participants attempted to make followed directly from the app ideas the students reported in the pre-survey, while other participants evidently realized Aptly's limitations and opted for a simpler app. For example, the participant whose original idea from the pre-survey was "an app that can read my mind and draw out what I'm picturing" abandoned this idea and started Task 3 with a completely different app idea, and the same is true for the participant whose original idea from the pre-survey was an app that "would be able to locate almost any item you had lost." Generally, participants seemed at least somewhat aware of Aptly's limitations after having interacted with it for the previous two tasks, and adjusted their app ideas to something there was a reasonable chance Aptly would be able to help them create yet was still a substantial app idea in most cases.

## 5.5 General observations

All participants answered "Agree" when asked whether Aptly was "easy to use." Nevertheless, seeing the way in which study participants interacted with the tool provided valuable insight

into some design issues that needed to be addressed, such as confusing button naming. Some higher-level observations were made as well:

First, participants displayed a general lack of patience. If an edit was not automatically implemented within a second or two, students assumed it was not going to work and began planning their next change.

When Aptly misunderstood participants' visions, some participants kept trying to utilize Aptly Editing over and over again with rephrased commands each time, and some gave up and changed it manually.

Generally, more experienced programmers tended to both 1) issue changes manually more often than non-programmers and 2) issue a higher number of commands than non-programmers. More experienced programmers also expressed a desire for more control, specifically being able to see the error message generated rather than just having the edit not work. When Aptly Editing fails to apply an edit because the Aptly code returned by GPT-4 cannot be parsed, a popup warning occurs, but the actual error message is not included. Concealing the error message from the user may help keep Aptly at a level more appropriate for beginners rather than scaring users with stack traces, but this decision seemed frustrating for those with programming experience who wanted a higher degree of control over their app creation process. Participants stated that it was frustrating to not know whether their command was unclear or whether something in their command referenced features that Aptly could not handle yet, so that they could choose how best to proceed. A possible path forward could be providing different modes for different experience levels, where the "Expert" mode could show the participant the error message produced including at least part of the stack trace, and "Beginner" mode could show a message like "Oops! An error occurred. Please try again with a different command."

Multiple participants assumed there would be an audio generation capability and were disappointed when Aptly couldn't generate sounds for them. This expectation might have been because participants were shown the image generation capability that utilizes DALL-E

[14] during the demonstration.

Most of the other limitations identified can be mitigated by a larger bank of examples that cover more MIT App Inventor components and features.

## 5.6 Success with incremental development

One particularly encouraging result observed during the study was the success participants were able to achieve by leveraging Aptly Editing's incremental development capability.

In addition to the examples mentioned earlier, one particularly striking example is as follows:

The participant who ended up with a creative solution to the scenario with a blind friend in need of a calculator app, namely the solution of tapping the screen a certain number of times to indicate numbers and operations and then having vibrations express the result, achieved this end result after starting with a much simpler and less capable app.

This participant began with the following description: "Make a calculator with buttons and a display that can do addition and subtraction."

Then, the participant issued the following sequence of commands to Aptly Editing:

1. "Under each of the two input text boxes add a voice button that hears what number a user says and enters that number into the text field"
2. "Make each button larger"
3. "Have a third button that hears one number, then hears either add or subtract, then hears a second number, after all three components are heard, then display the sum or difference of the two numbers in the top, then display the number in the display box too"
4. "Make two buttons at the top of the app. The first button represents number one, the second number two. For each tap, the number displayed on the button increases.

Under the two buttons are two buttons that either add or subtract the two numbers, and displays it under"

5. "Add number text boxes to the right of buttons 1 and 2, when a buttons is pressed, the number to the right of it goes up"

6. "Remove the two text boxes"

7. "Change button one's display text to zero at first and button 2's display text to zero at first too"

8. "Make each text and button bigger"

9. "Make each displayed element bigger. Also, when the result is displayed, vibrate the number of times of the result"

10. "Read the result when the results comes on"

11. "vibrate the number of times of the result when the result is displayed"

This series of commands demonstrates the participant's ability to recognize opportunities for improvement to the app and describe these changes to Aptly Editing. For example, the participant was able to recognize that larger buttons would be more effective, that the buttons should start out saying 0 if their value would be accumulated through taps, and that simply displaying the result on the screen may not be enough to help a visually impaired individual.

The participant was able to evaluate the current app after each incremental change, devise a plan for improving the app, utilize Aptly Editing to implement that plan, and evaluate the app again.

This participant's use of Aptly Editing mimics a workflow that might be utilized by professional developers, beginning with a minimal prototype and incrementally improving it, seeing the result at each step.

## 5.7 Changes in participants' attitudes towards programming and AI

Overall, participants had positive feedback regarding Aptly:

1. "This is a very helpful tool and it shows a way that AI can be used really well and it doesn't necessarily hinder your creativity, which is really nice."
2. "I think that AI is very useful and extremely beneficial when you need a problem solved that you can solve yourself, and it makes creating online resources (such as apps), more accessible for everyone."
3. "Aptly has a lot of potential to help people"
4. "I think [programming] is a lot more fun and easier with Aptly. As a beginner in programming I felt like I could also code an app based on the descriptions of the task."
5. "I think Aptly does a good job at ensuing interest into programming"
6. "I think Aptly has made programming a lot easier for me to think about."
7. "I am someone who really does not enjoy programming but I found aptly actually super interesting and fun to use."

However, other participants expressed concern regarding student programmers becoming too reliant on AI rather than understanding the logic themselves: "I think that it restricts the learning process, so if you want to actually learn how to code, using AI may not be the most helpful because it will just provide you with the immediate answers."

Another concern expressed was that AI will put "positions and employment opportunities at risk."

Evidently, while using an AI-powered tool encouraged the participants in terms of making programming more engaging, it did not dissipate students' fears about the potential negative

impact of AI in terms of students becoming overly reliant on it or workers losing their jobs as AI automation gains more momentum.

## 5.8 Conclusion

The overwhelmingly positive feedback, as well as promising results achieved by participants utilizing the tool to create and edit apps solving a range of tasks from specific recreation tasks to open-ended tasks, indicates Aptly Editing's potential to make programming even more accessible.

Aptly Editing's flexible approach, specifically its unique emphasis on being able to handle small, incremental changes in addition to one detailed description right from the start, facilitates usage by users with many different types of workflows and development styles.

Aptly Editing's flexibility in terms of its ability to contribute to different extents in the creative process as well as the technical implementation process also facilitates usage in a variety of contexts, and allows users to utilize Aptly as a true collaborator in the app development process in addition to just an assistant.

# References


[1] D. Wolber, H. Abelson, and M. Friedman, "Democratizing Computing with App Inventor," *GetMobile: Mobile Comp. and Comm.*, vol. 18, no. 4, pp. 53–58, Jan. 2015, ISSN: 2375-0529. DOI: 10.1145/2721914.2721935. URL: https://doi.org/10.1145/2721914.2721935.

[2] F. Shih, O. Seneviratne, I. Liccardi, E. Patton, P. Meier, and C. Castillo, "Democratizing mobile app development for disaster management," in *Joint Proceedings of the Workshop on AI Problems and Approaches for Intelligent Environments and Workshop on Semantic Cities*, ser. AIIP '13, Beijing, China: Association for Computing Machinery, 2013, pp. 39–42, ISBN: 9781450323468. DOI: 10.1145/2516911.2516915. URL: https://doi.org/10.1145/2516911.2516915.

[3] GitHub, *Introducing GitHub Copilot: your AI pair programmer*, 2021. URL: https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/.

[4] OpenAI, *Introducing ChatGPT*, 2022. URL: https://openai.com/index/chatgpt.

[5] Google, *Introducing Gemini: our largest and most capable AI model*, 2023. URL: https://blog.google/technology/ai/google-gemini-ai/#performance.

[6] E. W. Patton, D. Y. J. Kim, A. Granquist, R. Liu, A. Scott, J. Zamanova, and H. Abelson, *Aptly: Making Mobile Apps from Natural Language*, 2024. arXiv: 2405.00229 `[cs.HC]`.

[7] A. M. Granquist, D. Y. Kim, and E. W. Patton, "AI-Augmented Feature to Edit and Design Mobile Applications," in *Proceedings of the 25th International Conference on Mobile Human-Computer Interaction*, ser. MobileHCI '23 Companion, Athens, Greece: Association for Computing Machinery, 2023, ISBN: 9781450399241. DOI: 10.1145/3565066.3608248. URL: https://doi.org/10.1145/3565066.3608248.

[8] D. Kim, A. Granquist, E. Patton, M. Friedman, and H. Abelson, "Speak your mind: Introducing aptly, the software platform that turns ideas into working apps," in *ICERI2022 Proceedings*, ser. 15th annual International Conference of Education, Research and Innovation, Seville, Spain: IATED, Jul. 2022, pp. 1653–1660, ISBN: 978-84-09-45476-1. DOI: 10.21125/iceri.2022.0432. URL: https://doi.org/10.21125/iceri.2022.0432.

[9] OpenAI, *GPT-4 Technical Report*, 2024. arXiv: 2303.08774 `[cs.CL]`.

[10] OpenAI, *Introducing text and code embeddings*. URL: https://openai.com/index/introducing-text-and-code-embeddings.

[11] K. Zhang and D. Shasha, "Simple Fast Algorithms for the Editing Distance between Trees and Related Problems," *SIAM Journal on Computing*, vol. 18, no. 6, pp. 1245–1262, 1989. DOI: 10.1137/0218082. URL: https://doi.org/10.1137/0218082.

[12] *ZSS Module for Python.* URL: https://github.com/timtadh/zhang-shasha/.

[13] X. Deng and E. Patton, "Enabling Multi-User Computational Thinking with Collaborative Blocks Programming in MIT App Inventor," in *Proceedings of the 1st International Conference on Computational Thinking in Education*, Hong Kong, China, 13-15 July, 2017 2017.

[14] OpenAI, *DALL·E API now available in public beta*, 2022. URL: https://openai.com/index/dall-e-api-now-available-in-public-beta.