Group Collaboration with App Inventor

by

Xinyue Deng

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

June 2017

Author .....
Department of Electrical Engineering and Computer Science
May 18, 2017

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

Accepted by .....
Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Group Collaboration with App Inventor

by

Xinyue Deng

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Collaboration becomes increasingly important in programming as projects become more complex. With traditional text-based programming languages, programmers typically use a source code management system to manage the code, merge code from multiple authors, and optionally lock files for conflict-free editing. There is a limited corpus of work around collaborative editing of code in visual programming languages such as block-based programming. I present a collaborative programming environment to MIT App Inventor, a web-based visual platform for building Android applications with blocks, which enables many programmers to collaborate in real time on the same MIT App Inventor project. I design and implement three collaboration models to evaluate the efficacy of these different collaborative models for users of App Inventor so as to understand which approach best enables collaboration. Our results demonstrate that real-time programming decreases the completion time of a task, improves the interaction between users, and increases users' likeability towards collaborative programming. I anticipate that this new collaborative programming environment will change the way users use MIT App Inventor, and more curriculum based on the collaborative tools will be designed.

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

Acknowledgments

First, I would like to thank my advisor, Hal Abelson, for provide me with this opportunity to design and implement an important feature that will used by millions of users.

I could not have completed this project without Evan W Patton's support. He helped me to formalize the design of the collaborative programming environment, and figure out the steps I needed to finish the implementation. He also helped me design the user study.

I appreciate the generosity of Mike Tissenbaum for helping me finalize the metrics of my user study, and Ilaria Liccardi for advice on the design of user study.

I would not have a thesis without the participation of twenty MIT students. I have learned how people collaborate under different constrains by observing them. I thank each of the participants for their time.

I want to thank every undergraduate researchers in MIT App Inventor team, who participated in the testing of the system. Without their participation, I will not find bugs in the system and fix them before the user study.

My friend and fellow researcher, Natalie Lao, helped me edit my thesis and fixed my grammar mistakes.

My dog, Pocky, for being such a good accompany while I worked on my thesis.

Finally, a big thanks to the members of the MIT App Inventor developer team who answered my questions, reviewed my code, and collaborated with me on this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud-based collaborative technologies, such as Google Docs, have become an important part of how people work together in real time on all manner of content. However, real-time collaboration for programming is mostly in the research phase [5, 4, 6]. The typical collaboration strategy in text-based programming is for programmers to work separately, and then merge their changes through a version control system, such as Git. There are a few explorations on collaboration with visual programming tools [7, 8], like with Scratch and App Inventor, but there is still a long way to go before a usable platform can be released.

MIT App Inventor is a web application built for students and novice programmers to develop Android applications using a blocks-based programming language. Users can build the user interface of an application by dragging and dropping components, and write the logic of the application by connecting blocks. MIT App Inventor helps users to build Android applications without knowing the Java syntax and complex Android framework (More details about MIT App Inventor is in Chapter 3).

As of the time of this writing, MIT App Inventor has over 6 million users from 195 countries, and 164,000+ weekly active users build over 23 million Android applications [1]. As the number of users increases, and the complexity of projects grows, our team has received numerous requests for collaborative programming. In this thesis, I present a collaborative programming environment within MIT App Inventor that enables users to work on the same project simultaneously, and an experiment to

evaluate the efficacy of different collaborative strategies for novices and experts of App Inventor to understand which approach best enables collaboration.

1.1 Motivation

MIT App Inventor is widely used as an educational tool. It aims to help novice programmers mitigate the difficulties of programming for Android and make programming more accessible to young people. Studies show that collaborative programming, such as pair programming, improves the design quality, and reduces defects and improves the technical skills [22, 23]. It is important for MIT App Inventor to provide users a way to work collaboratively. Thus, young people and novice programmers would be able to learn how to work with others and build more complicated projects.

On the technical side, MIT App Inventor has millions of users and can support thousands of users online at the same time, so the collaboration environment adapted to it likewise needs to be highly scalable and structured. It has been an exciting challenge for me to work within such a large code base and to implement features that will be used by millions of users.

1.2 Use cases

Consider the following scenario: when a group of users wants to build a complex Android application with MIT App Inventor, they can divide the projects into multiple tasks and assign tasks to team members. In order to test the application, each user wants to know if their work is consistent with others and does not cause problems to others' tasks, so they want to see what other teammates are doing in the project and changes made by them. In addition, to speed up the progress, user wants to work on tasks in parallel with different computers.

With the current version of MIT App Inventor, users can only work on the project with one account, so they need to take turns to edit the project. A real-time collaborative programming environment within MIT App Inventor can solve this problem

by enabling users share project and see each other's change in real time.

1.3 Contribution

In my thesis, I present a collaborative programming environment within MIT App Inventor with three collaboration models and an experiment on the efficacy of different collaborative models. My goals were 1. to have a functional system inside App Inventor that allows users to work collaboratively. 2. to evaluate the efficacy of three different collaborative models for novices and experts of App Inventor so as to understand which approach best enables collaboration. Among three collaboration models, I found the real-time collaboration model is the most efficient because it decreases the completion time, improves the interaction between users and increases the users' positive attitude towards collaborative programming.

The following chapters explain each step of my project in more detail. In Chapter 2, I outline related work on collaboration in both text-based programming languages and visual programming tools. Chapter 3 introduces the basics of App Inventor and the technologies it was built on. In Chapter 4, I describe the design of the collaborative environment and the techniques I used to implement it, including modifications to MIT App Inventor. Chapter 5 contains the design, the metrics, and the results of an experiment on different collaborative strategies. In Chapter 6, I present a discussion on the efficacy of different collaborative strategies. Chapter 7 concludes my thesis with a section about future work.

1.4 A sample scenario using collaborative programming environment

Three users, Alice, Bob and Carol, want to build a simple Android application with MIT App Inventor. The application has three buttons, red, yellow and green, and a canvas. When user clicks on a button, the canvas changes its background color to the color shown on the button.

Using the collaborative programming environment, Alice creates a project called *ColorButton* and shares the project with Bob and Carol by clicking "Share Project" in the dropdown menu of "Projects" as shown in Figure 1-1.

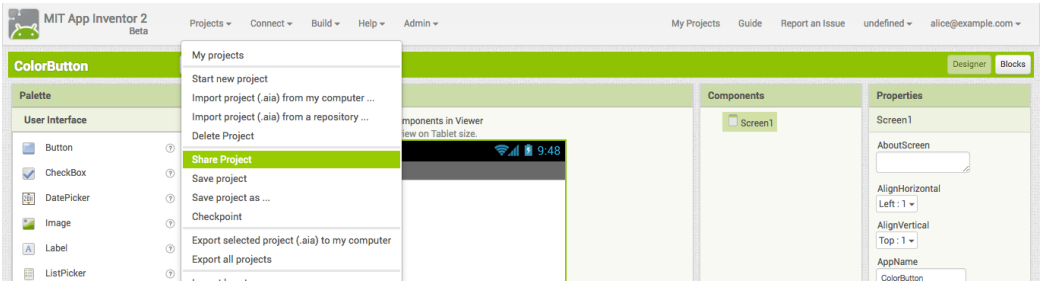


Figure 1-1: "Share project" menu item in the dropdown menu of "Projects"

When the dialogue pops up, Alice enters Bob's email address to share the project with Bob as shown in Figure 1-2

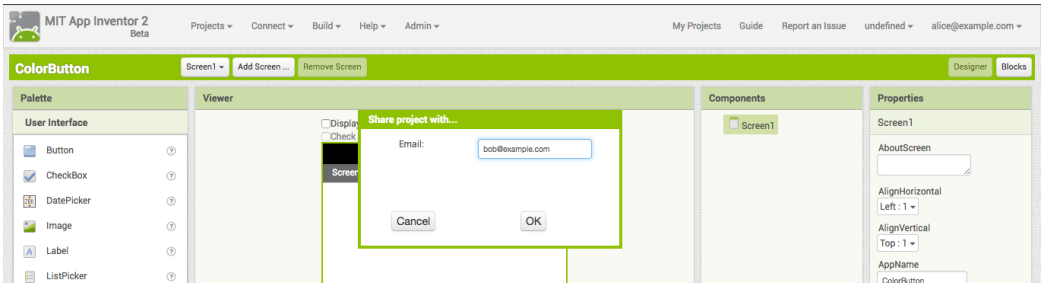


Figure 1-2: Alice shares project with Bob

Similarly, Alice shares the project with Carol as shown in Figure 1-3.

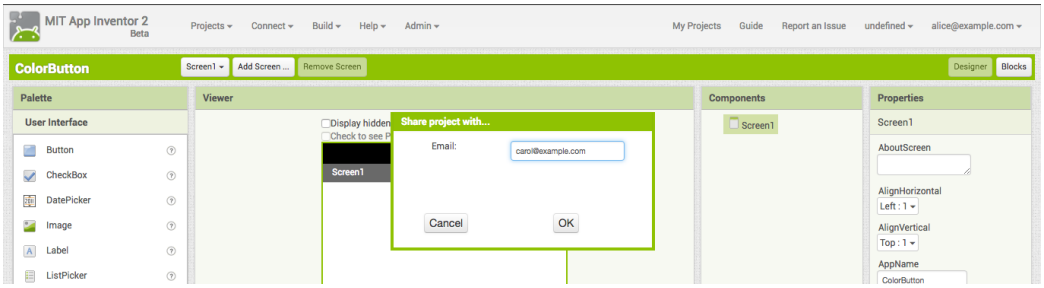


Figure 1-3: Alice shares project with Carol

After Alice shares the project, Bob and Carol can see *ColorButton* in their project listings. When other collaborators open the project, colored squares representing

other collaborators will appear next to the "Designer" button. When Bob and Carol open the project, Alice's project toolbar has two colored squares next to the "Designer" button as shown in the Figure 1-4. The red square with capitalized "B" represents Bob, and the blue square with capitalized "C" represents Carol. Since

Figure 1-4: The user interface of Alice after Bob and Carol open the project

the application needs three buttons and one canvas, Alice adds a canvas component and then splits tasks with Bob and Carol. They decide that Alice works on the red button, Bob works on the yellow button and Carol works on the green button. Each member adds a button below canvas, renames it and modifies the task to show the corresponding color. Figure 1-5 shows the *Designer* screen on Alice's computer.The red border surrounding the yellow button indicates that Bob is working on the yellow button, and the blue border surrounding the green button indicates that Carol is working on the green button. Then, Alice, Bob and Carol program the behavior

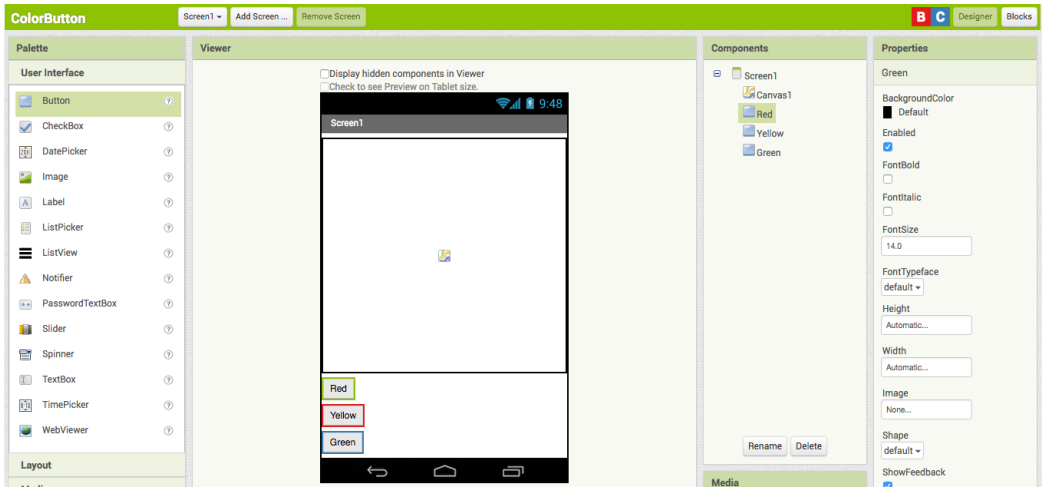


Figure 1-5: The user interface of *Designer* on Alice's computer.

Figure 1-6: The user interface of *Blocks* on Alice's computer.

of the button in the *Blocks*. When a button is clicked, the background color of the canvas changes to the corresponding color. Figure 1-6 shows the user interface of the *Blocks* on Alice's computer when three users work on the *Blocks*. The red border surrounding the blocks indicates that Bob is working on this set of blocks, and the blue border surrounding the blocks indicates that Carol is working on that set of blocks. The *ColorButton* application is successfully finished by Alice, Bob and Carol. Instead of working on the same machine, three users can modify the project in parallel using different computers with the collaborative programming environment within MIT App Inventor.

Chapter 2

Related Work

2.1 Collaboration in text-based programming

Modern programmers mostly collaborate through a version control system (VCS), where they work on different copies of the same code base on separate machines, and then merge their changes. It enables programmers to work on different files without writing over others' changes. In the case that two programmers edit the same file, the VCS will try to resolve the conflicts or report the conflicts to prompt the programmers to resolve them manually [2]. While the VCS provides teams with many benefits, it also introduces some difficulties in teamwork, such as isolating incoming changes from others that might affect one's current tasks. In 2015, Guzzi, Bacchelli, Riche and Van Deursen interviewed several professional software developers, and found that developers had issues about dealing with others' changes, such as when new changes changed the dependencies or introduced bugs in the tasks they were working on. Since developers lacked the same information about each other's changes, it was hard to locate the bugs in the new changes. Therefore, Guzzi et al. developed a new extension to IDE, called *Bullevue*, which enabled developers to see the history of changes of each file visually in the editor, and in which developers are able to reverse changes based on the sub-file level [3].

Other than VCS, real-time collaboration in text-based programming has remained mostly confined to research settings. Boyer et al. developed a plugin, *Ripple* for

Eclipse, an integrated development environment (IDE) to enable students to collaborate distributively for educational purposes. *Ripple* provides a synchronized programming view and a chat channel for programmers. Developers can see others' change in real time, and they can use the chat channel to communicate [4].

In 2011, Goldman et al. presented *Collabode*, a web-based Java IDE, to support real-time collaborative programming. One of the main challenges is to make sure the code is always error-free, so that any collaborator can compile and run their code anytime. *Collabode* solved this problem by only sharing changes if they can compile successfully [5].

In 2016, Ghorashi et al. built a web-based editor for web programming called *Jimbo*, which supports synchronous and asynchronous collaboration. It deployed operational transforms to achieve eventual consistency with multiple users working on the same code. It enabled developers to audio chat and text chat to communicate, and provided a notification system to inform developers about code changes of certain files. To test the code, it offered a live preview of the website when receiving new changes [6].

2.2 Collaboration in blocks-based programming

blocks-based programming languages allow developers to program by dragging and dropping blocks. They are designed to lower the barriers to programming for young people and novice programmers, because they do not need to worry about the syntax.

AlgoBlock is a tangible programming tool designed for primary and secondary students to program with physical blocks. Using this tangible programming tool, students collaborate in a shared workspace, and can monitor others' work by their body movements. Different from conventional programming with a personal input, such as keyboard and mouse, AlgoBlock made the process of programming open, so that it is easier for students to collaborate and interact [7].

In his master's thesis in 2015, McKinsey presented a prototype of remote pair programming, called *Turtle Tango*, in open online courses with blocks-based pro-

gramming. The prototype allows developers to program a turtle to draw on a pane with blocks collaboratively by receiving others' changes simultaneously. Users can also communicate via video chat. In the pilot study, McKinsey found that, by receiving others' changes in real time, developers finished the same tasks in less completion time than without receiving it [8].

There are also other blocks-based programming tools used widely, such as *Scratch* [9], and *App Inventor* [10]. However, they do not support collaborative programming in their current versions.

Chapter 3

Introduction to App Inventor

3.1 Overview

MIT App Inventor is a web application for creating Android apps with blocks-based programming language. It was developed by Google's Mark Friedman and MIT professor Hal Abelson while Abelson was on sabbatical at Google. The major goal of MIT App Inventor is to make Android programming easier to access for novices. Instead of needing to understand the Android framework and use complex Java syntax, novice programmers can build an Android application with visual drag-and-drop blocks within one hour or less [1].

MIT App Inventor consists of two editors: *Designer* and *Blocks*. In *Designer*, developers can design the user interface of an application by dragging and dropping components onto the screen. There are two types of components: visible components, such as Button and Label, are shown inside a mock phone screen in the editor; non-visible components, such as Camera and TinyDB, are shown in a list under the mock phone screen. Each type of component has a set of properties that developers can configure, and a set of blocks that developers can use to control the behavior of a component. Figure 3-1 shows an example of *Designer* of PaintPot, an application that allows a user to draw on an image with three different colors. To access the blocks, developers can toggle from *Designer* to *Blocks*.

Developers use *Blocks* to control the behavior of the application. The blocks can

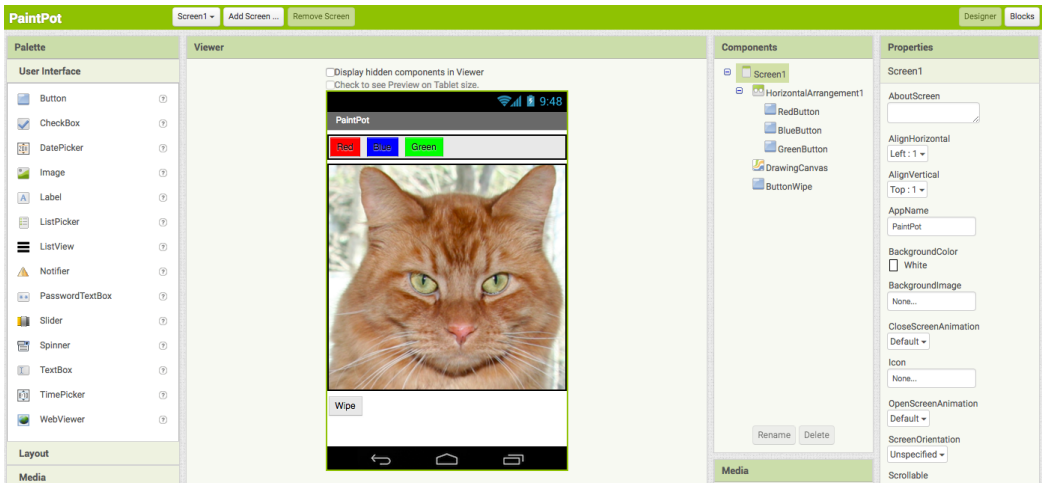


Figure 3-1: Designer editor for PaintPot, an application that allows a user to draw on an image with three different colors.

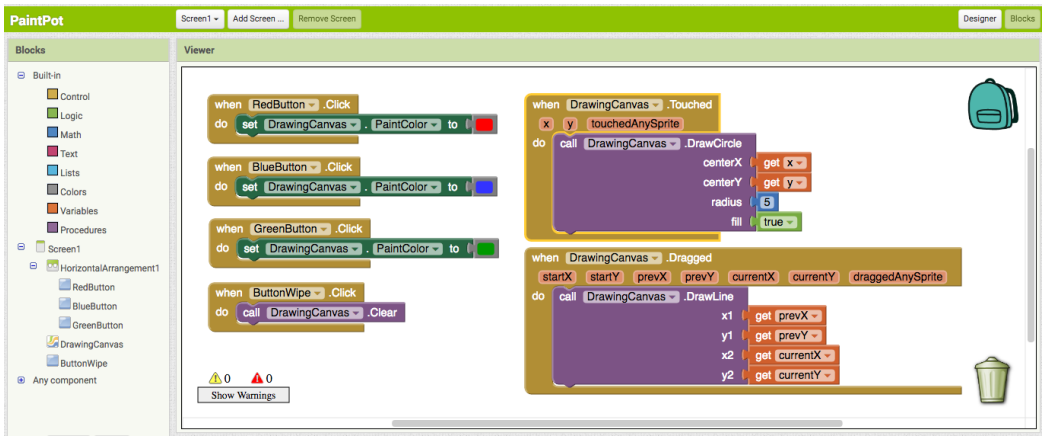


Figure 3-2: Blocks editor for PaintPot, an application that allows a user to draw on an image with three different colors.

be dragged and dropped onto the workspace, and connected with other blocks to form a complete logic. There are built-in blocks and component-specific blocks. The built-in blocks contain logic common to many programming languages, such as lists, conditions, and loops. The component-specific blocks contain the specific functions related to the type of the component. On the workspace, developers can move a set of blocks to any location by dragging them. They can expand and collapse a set of blocks, so that they can customize the view of the workspace to work on a certain set of blocks. In addition, they can enable and disable blocks. By disabling a set of blocks, this set will not be executed by the application. Figure 3-2 shows *Blocks* of

PaintPot.

3.2 Technology

MIT App Inventor uses a client-server architecture. The client side is responsible for the user interface, and the server side is in charge of services and database. The client runs inside a web browser and is built on top of Google Website Toolkit (GWT) and Blockly. The server runs on Google App Engine (GAE) using the GWT server as a GAE service and a third-party datastore API called Objectify. Figure 3-3 shows the relationship between the components of the MIT App Inventor system.

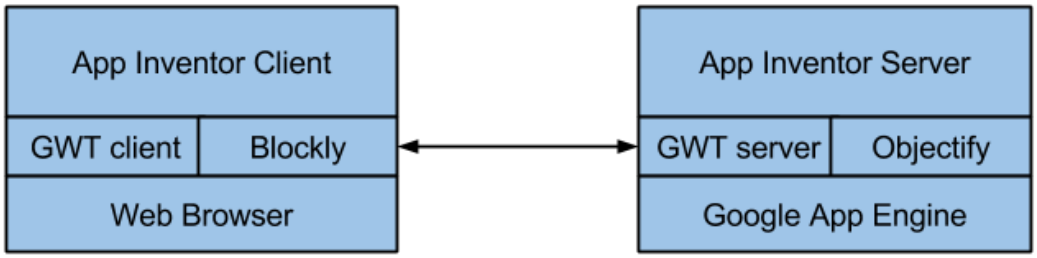


Figure 3-3: MIT App Inventor system architecture

Inside MIT App Inventor, users are identified by their email addresses, and the system will assign each user a unique identifier in the database. With the current implementation, each user can only have one session ongoing, which means the user can only open one MIT App Inventor instance in their browser.

3.2.1 Google Website Toolkit

Google Website Toolkit (GWT) is an open source tool for building web applications with Java. It compiles and optimizes Java code to JavaScript code for different kinds of browsers. On the client side, GWT provides components for the user interface, such as buttons and text boxes, and handles mouse and keyboard events. On the server side, it integrates AJAX to support communication between client and server. The GWT client runs inside the web browser, and the GWT server can run on the Google App Engine. MIT App Inventor deploys both the GWT client and GWT

server, and uses the GWT remote procedure call framework to exchange Java objects between client and server. Another important feature of the GWT is that it allows developers to integrate native JavaScript code into Java code with its JavaScript Native Interface (JSNI) feature. Therefore, it is easy for developers to integrate a third-party JavaScript library or an existing JavaScript code into a GWT application [11].

3.2.2 Blockly

Blockly is an open-source JavaScript library to create a visual programming editor for both the web and mobile apps. It is being developed and open sourced by Google. Blockly provides a workspace and a toolbox. The workspace is the editor where users can drag and drop blocks, and the toolbox is the drawer containing available blocks. Developers can define the types of blocks in the toolbox, and they can also customize new types of blocks. To generate actual code from blocks, developers can use the built-in generator or define their own generator [12].

In Blockly, there is an event system to catch any changes in the workspace. The event system abstracts users' actions into JavaScript objects, and Blockly implements multiple event listeners to perform different actions when receiving an event. When we implement the collaborative programming environment, these events are used to transfer changes between users. In Blockly, they define the following events:

- Blockly.Events.CREATE:
  This event is fired when user creates a new block in the workspace.

- Blockly.Events.DELETE:
  Fired when user deletes a block in the workspace.

- Blockly.Events.MOVE:
  Fired when user moves a block in the workspace to a new location, or attaches it to another block.

- Blockly.Events.CHANGE:

Fired when user changes a property of the block, such as a field of the block or disabling the block.

- Blockly.Events.UI:
  Fired when user changes the user interface of the block. For example, user selects the block.

3.2.3 Data Store

Since MIT App Inventor runs on the GAE, it stores data in Google Cloud Storage (GSC), and it has a relational database using Objectify. When a new user signs up, the server creates a row for the user in the relational database. When the user creates a project, the server creates a row for the project, and another row that links the user and the project in the relational database. GSC only stores the files related to the projects. For each project, MIT App Inventor stores two files. One file stores information about all components in *Designer* with JSON format, and the other file stores information about all blocks in *Blocks* with XML format. The server saves projects every five seconds after it detects new changes, and it generates two new files corresponding to *Designer* and *Blocks* and replaces the old files of this project in the storage. For each user, project, component, and block, the server assigns them a unique identifier to distinguish them.

3.3 AIMerger

With the current version of MIT App Inventor, there are two approaches for users to collaborate on the same project. The first approach is that different users edit the same project by turns with the same email account. The second approach is remixing, in which one user exports the project as an AIA file after finishing his part, and then another user imports the AIA file to her account and works on her part of the project. Neither approach allows users to work on a project at the same time.

In 2012, Feeney created AIMerger, a project merger tool for MIT App Inventor,

in a master's thesis. It allows developers to merge two App Inventor projects into one project, and developers can select the screens and assets of each project they want to merge. Developers can collaborate with this tool by dividing tasks into different screens and then merging them. However, not all tasks of a project can be easily divided by screens, and collaborators lack the information about what partners are working during the collaboration process. In addition, because they do not know other parts of the application before merging, there could be bugs and problems after merging the project. Then, they only have one project, so they cannot collaborate any more without dividing up the project again [13].

Chapter 4

System Design and Implementation

In this chapter, we present our design and implementation of the collaborative programming environment for MIT App Inventor.

4.1 Real-time collaboration vs. version control systems

As mentioned in chapter 2, there are two major approaches for collaborative programming: real-time collaboration and version control systems. The traditional approach, version control systems, does not transfer well to blocks-based programming in MIT App Inventor for these reasons.The MIT App Inventor server generates one file for each editor of the project, every change in the editor results in a new file, which introduces conflicts when using a version control system. For example, if one developer detaches a set of blocks from its parent block and attaches this set to a new block, the XML file it produces will change significantly. To resolve the conflict, developers need to manually choose which part of the file they want to keep, which makes collaboration complex and laborious. In addition, unlike traditional text-based programming language, blocks-based programming language has a two-dimensional workspace, which introduces more difficulty in showing conflicts and the history of changes made by individual developers.

In contrast, real-time collaboration does not require developers to resolve conflicts manually, and it is easier to show others' changes simultaneously in the workspace. This led us to develop a real-time collaboration model to approach the collaborative programming inside MIT App Inventor.

4.2 Design Requirements

MIT App Inventor is designed to make programming more accessible for novice programmers and young people, and it is used mostly in introductory computer science courses. Therefore, our collaborative programming environment is designed for group course projects of two-to-four students in middle schools, high schools, and colleges.

The current version of MIT App Inventor only allows one user per project, but we want developers to be able to share projects with others in the collaborative programming environment. While using real-time collaboration, developers should be able to see others' changes simultaneously.

Another important design aspect of a collaborative programming environment is supporting user awareness. There has been much research about user awareness in the distributed work groups, and many techniques has been proposed [14]. According to Dourish, "Awareness involves knowing who is 'around', what activities are occurring, who is talking with whom; it provides a view of one another in the daily work environments. Awareness may lead to informal interactions, spontaneous connections, and the development of shared cultures" [15]. User awareness plays a crucial role in the collaboration, because it provides context for individual work with the group progress. Therefore, the collaborative programming environment inside MIT App Inventor should provide collaborators with enough information about what their partners are doing in the same project.

In summary, our collaborative programming environment should fulfill the following requirements:

1. Users can share projects with others by using their email addresses.

2. Users can work on the same project at the same time with different accounts. The projects in each user's account are always identical. When users open the project, they can see others' changes in real time.

3. While opening the project, users can know who is currently working on the project, and which part of the project their collaborators are working on.

4.3 User Interface Design

In order to support sharing projects and user awareness, we made a few modifications to the user interface of current MIT App Inventor.

4.3.1 Share Project

We added a menu item, called "Share Project" in the drop-down menu of "Projects" in the toolbar. After users click the menu item, a dialogue will pop up for inputting other collaborator's email address. Users can share the project with multiple collaborators, but they can only input one email address at one time. Figure 4-1 and Figure 4-2 show the menu item and the dialogue for sharing projects. After users share the

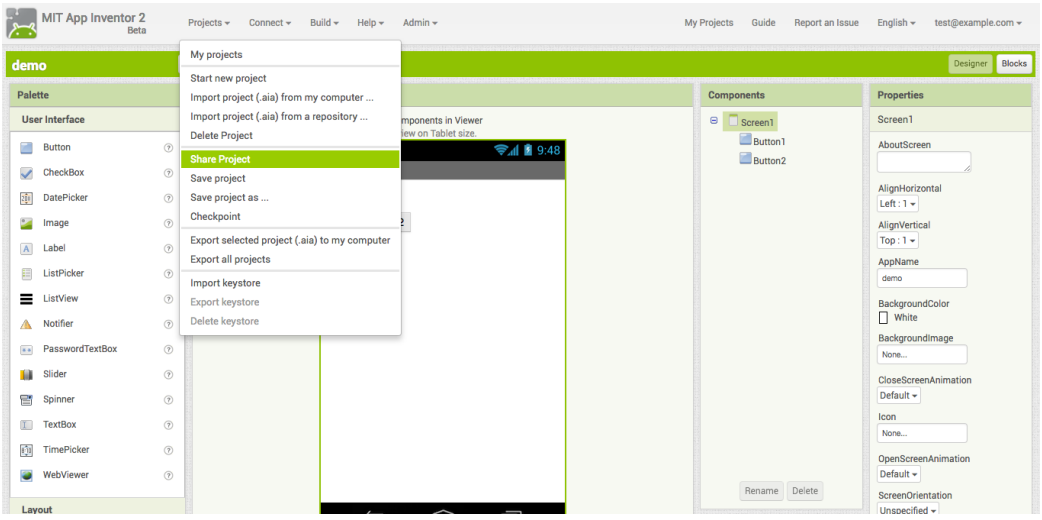


Figure 4-1: The menu item to share projects in the drop-down menu of Projects

project, their collaborators can find the project in the project listings.

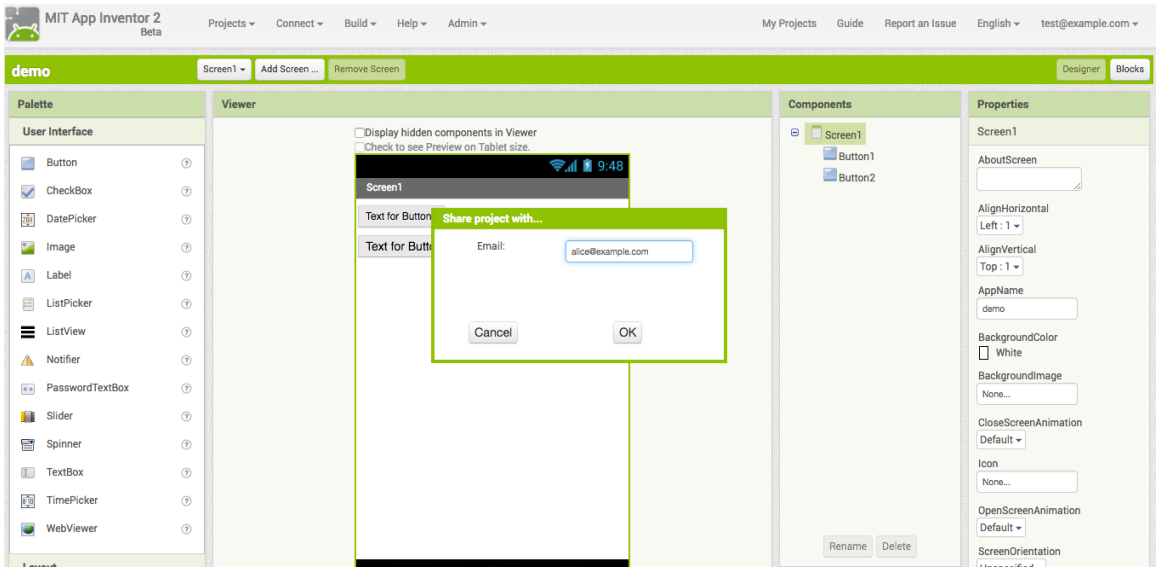


Figure 4-2: The dialogue to share projects with the collaborator's email address

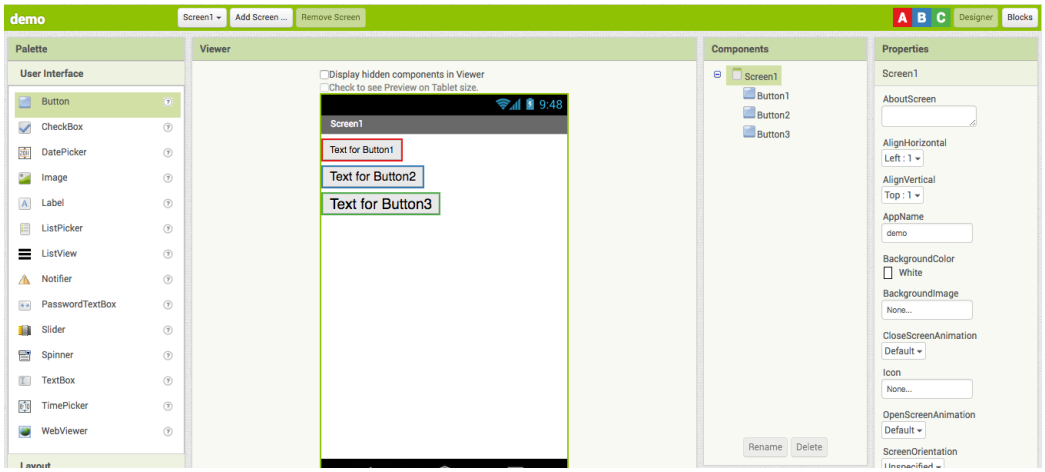
4.3.2 User Awareness

In order to support user awareness, we approached this with the similar techniques that Google Docs uses [16].

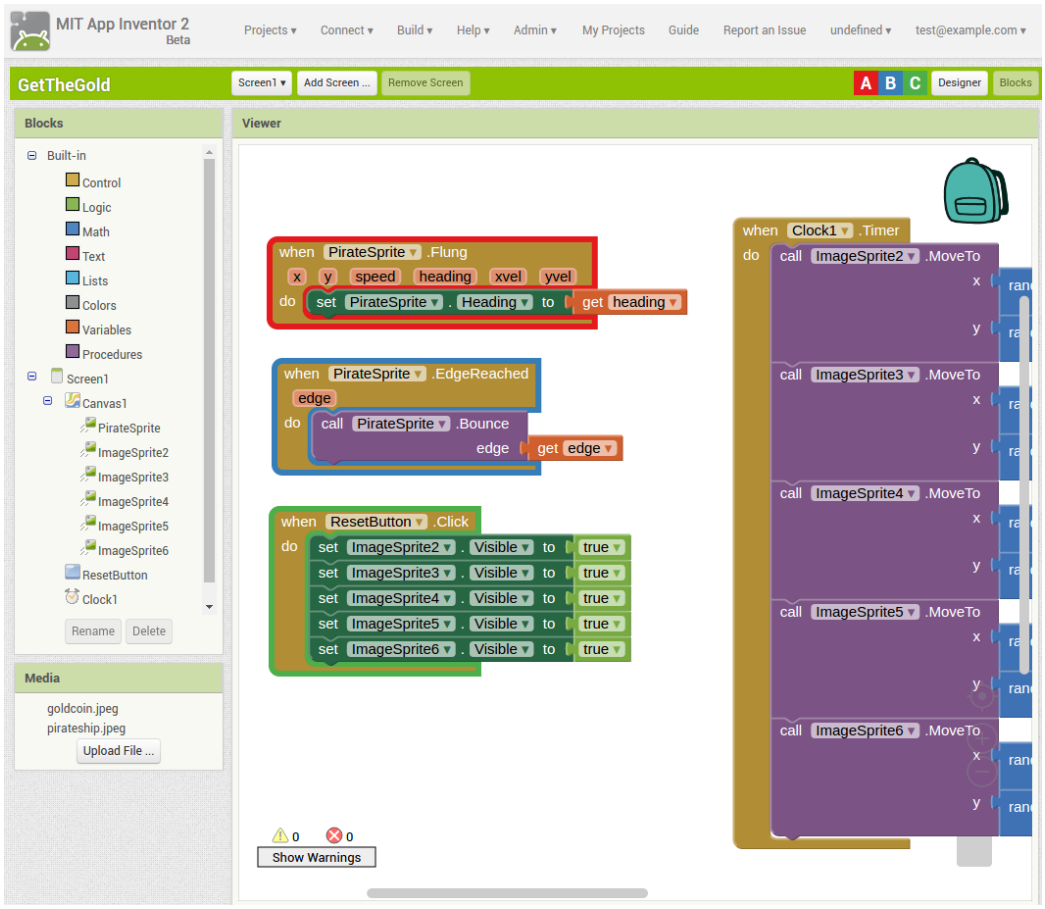
When a user opens a project, he will be assigned a color. If there are other users who have already opened the project, the new user will be represented as a colored square next to *Designer* button. There can be many colored squares if multiple collaborators open the project. Inside the colored square, there is the capitalized first letter of new user's email address. If the user hovers his mouse over the colored square, it shows the full email address of the collaborator whom the square represents. If the user closes the project, the colored square representing him will disappear from his collaborators' screen. In order to notify the user what his collaborators are working on, we surround the component and the block with the same color as the colored square that represents the collaborator. Figure 4-3 shows an example of showing user awareness in both *Designer* and *Blocks*.

In addition, users can see each others' changes simultaneously in the same project. There are following scenarios in the MIT App Inventor:

1. When two users work on different screens, their changes will not be shown until one user opens the other screen.



(a) An example of showing user awareness in *Designer*



(b) An example of showing user awareness in *Blocks*

Figure 4-3: An example of showing user awareness in MIT App Inventor

2. When two users work on the same screen, and they work on the same editor, they can see the others' change immediately on the editor.

3. When two users work on the same screen, but they work on different editors, the user on *Blocks* can see the new component added by the user on *Designer*, and he can add blocks related to the new component. When the user on *Designer* removes a component, the user on *Blocks* will find that blocks related to that component are deleted.

To reduce the number of requests sending to the server while maintaining the consistency in case of network failure, projects are reloaded from the server whenever users open or refresh a project. While users open a project, changes to all screens of that project will be synchronized, so users do not need to request data from the sever frequently.

4.4 Modification to MIT App Inventor

The current implementation of the server of MIT App Inventor is designed for single user, so we made several modifications to it to integrate the collaborative programming environment. We modified schemes in the database to memorize collaboration-related data, and we implemented an event system to facilitate how messages are transferred between users.

4.4.1 Data store

A new entry we added to the project data is the permission. When users share a project with a collaborator, the collaborator's identifier is added into the permission of the project, and then a link to the project is created in the collaborator's profile. Then, when users refresh their project listing, the shared project will appear. In addition, the previous implementation in MIT App Inventor only allows the project owner to upload and delete files of a project, so we modified it to that all users in a project's permission could upload and delete files of the project.

4.4.2 Event System

Since MIT App Inventor was designed for single user, and it makes the original implementation too complex to support transferring messages between users. We refactored the original code, and implemented an event system in the *Designer* editor to facilitate the message transfer for collaboration. The Blockly library has already provided an event system as discussed in Section 3.2.2, so we used it to transfer changes related to *Blocks*.

Event handling follows the observer design pattern, where one object maintains a list of observers, and notifies them when a state changes [17]. In our case, the object is *Designer*, and the observers are event listeners. When users perform an action, such as dragging a component onto the mock phone screen or changing a property of a component, the corresponding event will be constructed and fired. When the event is fired, it will notify the event listeners in *Designer*, and then the listeners will execute the corresponding action. To clarify our implementation, we created an abstract class *ComponentEvent* containing the common methods used for all events in *Designer*. All event classes associated with *Designer* extend *ComponentEvent* and customize their own functions. To transfer data between users over internet, the events need to be encoded in JSON format. Therefore, each event has the following functions:

1. *toJson()*: convert the Java event object to a JSON object.

2. *fromJson()*: convert the JSON object to a Java event object.

In addition, since MIT App Inventor uses both GWT and JavaScript in the front-end, we created the event objects in JavaScript and its mirror Java class in the GWT, so that the system can easily exchange information between GWT and JavaScript. Table 4.1 summaries the properties and functionality of all events we created for *Designer* and *Blocks*. We added an event of selecting blocks to change the border color of the blocks when a user selects it. Although Blockly provides an event for selecting blocks, it does not perform the action we needed.

Event class	Property	When event is fired
CreateComponent	• User ID • Component ID • Component type	Create a component instance with the given component type and component ID
DeleteComponent	• User ID • Component ID	Delete the component with the given component ID from the *Designer* editor.
MoveComponent	• User ID • Component ID • Parent component • Index	Move the component with the given component ID to the parent component at the given index.
SelectComponent	• User ID • Component ID • Selected (boolean)	Select or deselect the component with the given component ID depending on the given selected value. If the component is selected, set the component's border color as the same color of the given user ID.

ComponentProperty	• User ID • Component ID • Property name • Property value	Change the current value of the property of the component with the given component ID to the given property value.
SelectBlock	• User ID • Block ID • Selected (boolean)	Select or deselect the block with the given block ID depending on the given selected value. If the block is selected, set the block's border color as the same color of the given user ID.

Table 4.1: Summary of the properties and functionality of events related to *Designer* editor

4.5 Collaboration Server

In order to see others' changes simultaneously without reloading the project, the changes should be transferred to collaborators' clients immediately after submitting the changes. Therefore, we implement an independent NodeJS server to handle the message transfer between clients. The reason that we did not build the collaboration unit on the original server of MIT App Inventor is that it is harder to implement message transfer with Java framework, and it is easier to maintain the code by separating the collaboration and main services. Inside the NodeJS server, we used Socket.io [18] to manage the communication between clients and servers, and the Pub/Sub message system inside Redis [19] to manage the communication between clients.

4.5.1 Publish-subscribe pattern

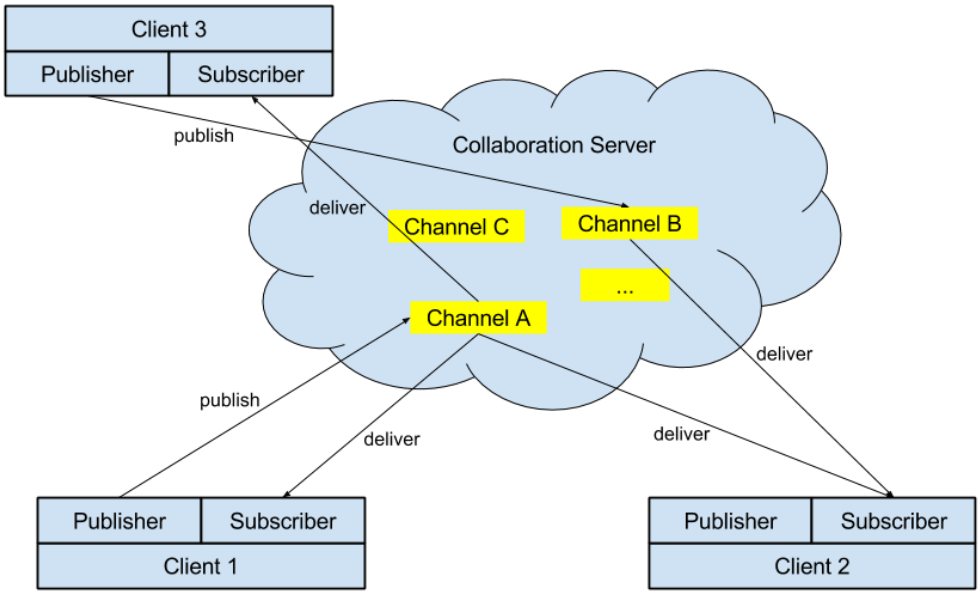


Figure 4-4: The publish-subscribe pattern inside MIT App Inventor

Publish-subscribe pattern is a messaging pattern widely used in peer-to-peer applications, which the publisher, the sender of the messages, does not send the messages directly to the specific subscribers, the receivers of the messages, but instead send it to a channel or an entity, where subscribers of the channel will receive the messages automatically. In this messaging pattern, the publishers and the subscribers can communicate without the knowledge of each other [20].

In our collaborative environment, each MIT App Inventor client is both publisher and subscriber. When users log in, they will subscribe to multiple channels as needed. When they perform actions, they will send the message encapsulating the related events to the collaboration server, and tell the server which channel they want to publish the message on. Then, the clients that subscribe the same channel will receive the message, transfer the message to the event, and the perform the corresponding action. Figure 4-4 shows how clients are communicated with each other with the publish-subscribe messaging pattern inside MIT App Inventor. With

the publish-subscribe pattern, developers can receive changes about the project after their collaborators submit the changes.

4.5.2 Channels

Each MIT App Inventor client will publish and subscribe to three kinds of channels:

1. User channel: The user channel is specified by the user's email address. Each client subscribes to only one user channel, which is his own email address. When users share a project, they publish the project and their information to the collaborator's user channel. Therefore, the collaborator will be notified that a user shares a project with him, and that project will appear in his project listing.

2. Project channel: Project channel is specified by project id. When users open a project, they will subscribe to that project channel. This channel is used for project-level messages. There are three kinds of project-level messages:

   - User join: When other users open the project, they will publish their information to the project channel, so that its colored square will appear on other collaborators' screen.
   - User leave: When other users close the project, they will publish their information to the project channel, so that its colored square will disappear on other collaborators' screen.
   - Media file: When users upload or delete a media file of the project, they will publish the information about the file to the project channel, so that the related file will appear or disappear from other collaborators' projects.

3. Screen channel: The screen channel is specified as combination of the project id and the screen name. Users can subscribe to multiple screen channels at the same time, because there could be multiple screens in a project. This channel is used to publish changes about components and blocks of this screen. When users perform an action on the editors, such as adding a component or dragging

a block, the related events as we discussed in section 4.4.2 and users' information will be published to the screen channel. Then, other collaborators can extract events from the received message and run the event on their side, so that they can see others' changes.

4.5.3 Collaboration Event Listener

To publish changes of a screen to the channel, we implemented a class called *CollaborationManager*, which manages the connection between the MIT App Inventor client and the collaboration server, and acts as an event listener of the *Designer*. Whenever an event is fired, it will be invoked and publish the corresponding events to the screen channel. Inside this class, there are following functions:

- *onComponentPropertyChanged()*: Called when a property of a component is changed, publish a *ComponentProperty* event to the channel.
- *onComponentRemoved()*: Called when a component is removed, publish a *DeleteComponent* event to the channel.
- *onComponentAdded()*: Called when a component is added, publish a *CreateComponent* event to the channel.
- *onComponentRenamed()*: Called when the name of a component is changed, publish a *ComponentProperty* event to the channel.
- *onComponentMoved()*: Called when a component is moved, publish a *MoveComponent* event to the channel.
- *onComponentSelectionChange()*:Called when a component is selected or deselected, publish a *SelectComponent* event to the channel.

On the *Blocks*, we registered a new event listener to the workspace, which publishes the Blockly events to the channel when the workspace receives a change.

In summary, with changes in the data store and user interface and the implementation of the event system and collaboration server, users can share projects with

others and see others' changes in the real time, and they can know who are currently editing the project.

Chapter 5

Experiment

In this chapter, I present an experiment with the collaborative programming environment for MIT App Inventor. In order to find out which approach best enables collaboration, I made some modifications to support different collaborative schemes to study how people collaborate under different constraints.

5.1 Collaboration Models

The design of a collaborative environment wants to satisfy both the needs of an individual and the needs of a group, but there is always the tradeoff between individual power and the group awareness. In order to maintain the group awareness, such that all collaborators have the same level of information about the project, the collaborative environment sometimes needs to limit the individual control over the project [21]. For example, with the real-time collaboration system introduced in Chapter 4, one can delete a component while others are working on that component, which could introduce conflicts into the collaboration. As the collaboration system I designed is the first collaborative programming environment for MIT App Inventor, it is hard to decide how much individual power user should have in order to achieve the greatest efficacy of collaboration. Therefore, I design three different collaboration models to test which approach best enables the collaboration between multiple users: 1. Project-level collaboration model 2. Component-level collaboration model 3. Real-time

collaboration model

Project-level collaboration model allows only one user to edit the project at any time. Although the changes are shown on every collaborator's computer, we expect to see users work on the same screen if they are co-located. More details about project-level collaboration model are introduced in Section 5.1.1. Component-level collaboration model allows users to edit different components or blocks, but two users cannot modify the same component or blocks at the same time. We expect users split the tasks based on the components and blocks. More details about component-level collaboration model are introduced in Section 5.1.2. The real-time collaboration model allows users to edit any components or blocks at anytime, and it has the same system introduced in Chapter 4. Users can have different collaboration strategies with the real-time collaboration model. For example, they can split tasks first and work separately, or they can discuss solutions and work together.

We expect that users collaborate differently with different collaboration models. Take the scenario discussed in Section 1.4 as an example, with project-level collaboration model, Alice will edit the project to add buttons and their logic. Bob and Carol will watch her and offer her advice. With the component-level collaboration model, they will split the tasks based on buttons, and after they finish, they will debug together. With the real-time collaboration model, they can split the tasks based on buttons, but they can correct each other directly if they find a bug.

5.1.1 Project-level Collaboration Model

Pair programming, where two programmers work on the same project side by side on one computer, has been a popular collaborative strategy in the last decades. In pair programming, one programmer is the "Driver", who edits the project and controls the keyboard and the mouse, and the other programmer is the "Observer", who views the code and looks for any logic flaws and syntax errors. Two programmers switch roles after a designated period of time. While pair programming takes more time to finish the project, it increases the design qualities, reduces defects and improves the technical skills of both programmers [22, 23].

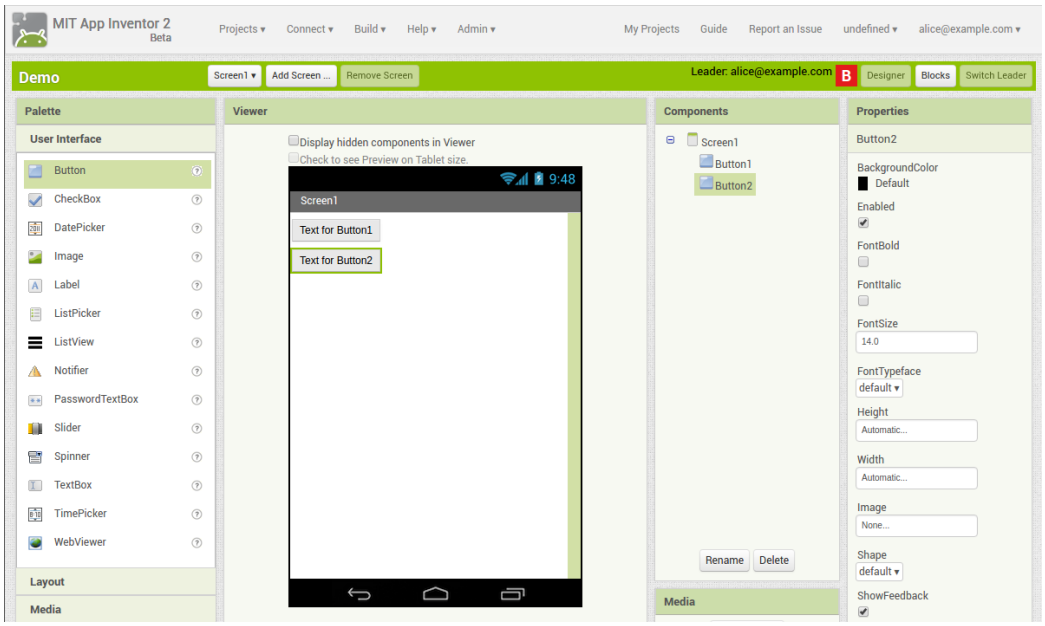


Figure 5-1: User Interface of the leader of a project with project-level collaboration model

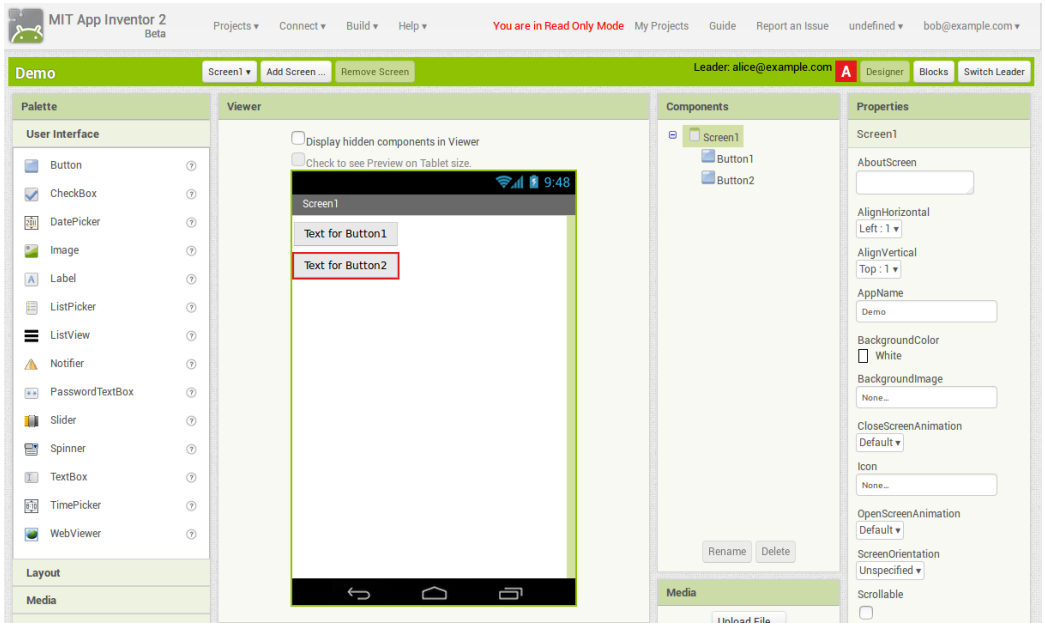


Figure 5-2: User Interface of the non-leader of a project with project-level collaboration model

We took pair programming as an inspiration, and designed the project-level collaboration model, where only one user can edit the project at any given time, but instead of working on the same machine, users can view the project on different

machines, and see the changes to the project simultaneously. With project-level collaboration model, each project has a leader, and only leader can edit the project, and other collaborators are in the read-only mode. Figure 5-1 and Figure 5-2 shows the user interface of the leader and non-leader with project-level collaboration model. Users can click the "Switch Leader" button to switch the leadership. After switching leadership, users can edit the project, and other collaborators will be in the read-only mode, and the text showing the current leader will be changed to the corresponding user's email address. When non-leaders try to edit the project, they will receive a

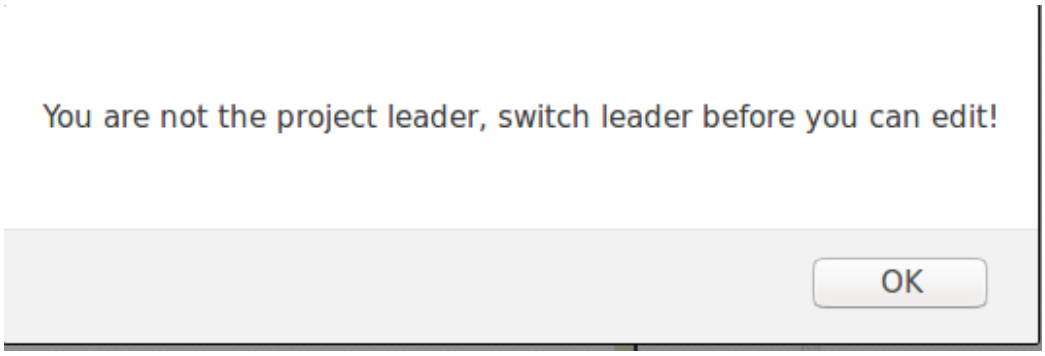


Figure 5-3: Error message when non-leaders edit the project

pop-up error message showing that they are unable to edit the project, and they need to be the leader to edit the project. Figure 5-3 shows the error message dialogue.

5.1.2 Component-level Collaboration Model

While project-level collaboration model only allows one user to edit the project at any given time, the component-level collaboration model provides users with more power over the project. The projects in MIT App Inventor consist of components and blocks, so with component-level collaboration model, a component or a set of blocks can be modified by only one user at any given time. Users obtain the lock for the component or a set of blocks when they select them, and release the lock when they deselect them. When users select a component, other collaborators cannot modify this component, but they can modify the blocks associated with this component, which allows users to split tasks by *Designer* and *Blocks*. When users select a block,

the entire stack of block's parents are also locked. Users cannot modify components or blocks locked by other collaborators. After users obtain the lock of a component or a block, an event will be sent to other collaborators to notify them the component or blocked is locked by the users.

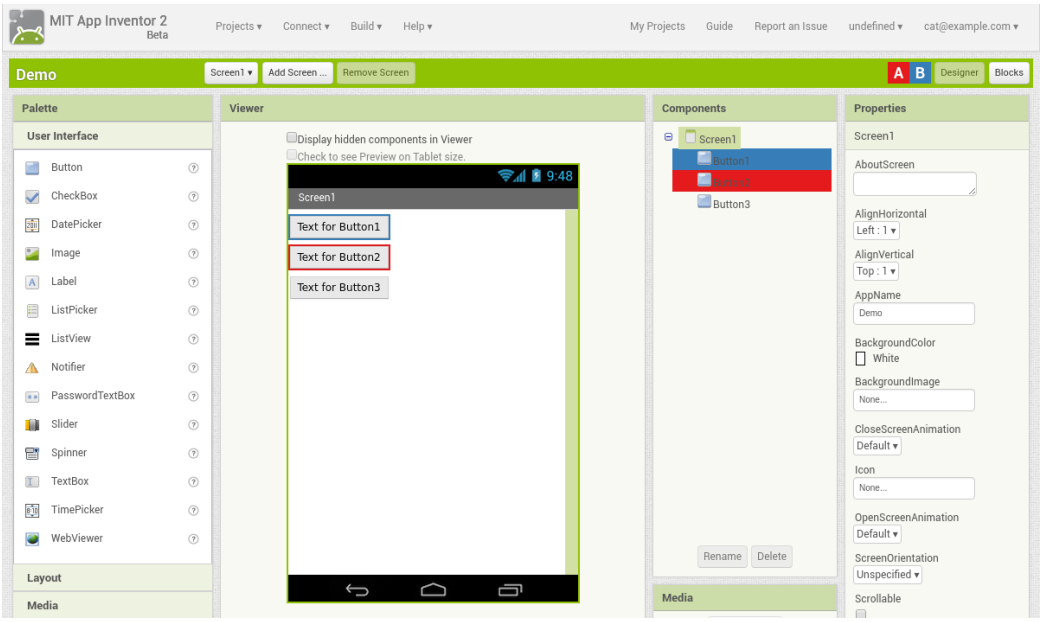


Figure 5-4: User Interface of the *Designer* with component-level collaboration model

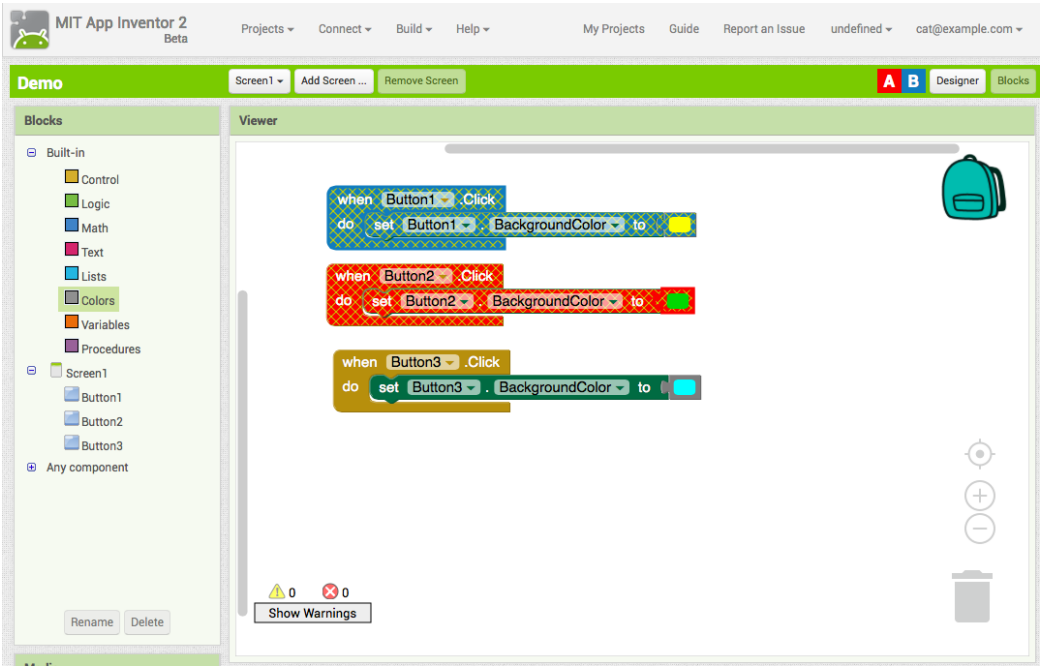


Figure 5-5: User Interface of the *Blocks* with component-level collaboration model

In *Designer*, users can see that the background colors of the components in the right side panel reflect the colors of other collaborators who currently hold the lock for the components. Figure 5-4 shows an example of the *Designer* in the component-level collaboration model, where User A locks Button 2, and User B locks the Button 1.

In *Blocks*, when users select or deselect a block, the entire hierarchy of this block, including its parent blocks and child blocks, will be locked or unlocked respectively. When blocks are locked, their background colors reflect the colors of the collaborators who currently hold the lock for the blocks. Figure 5-5 shows an example of *Blocks* in the component-level collaboration model, where User A locks the blocks "When Button 2 is clicked", and User B locks the blocks "When Button 1 is clicked".

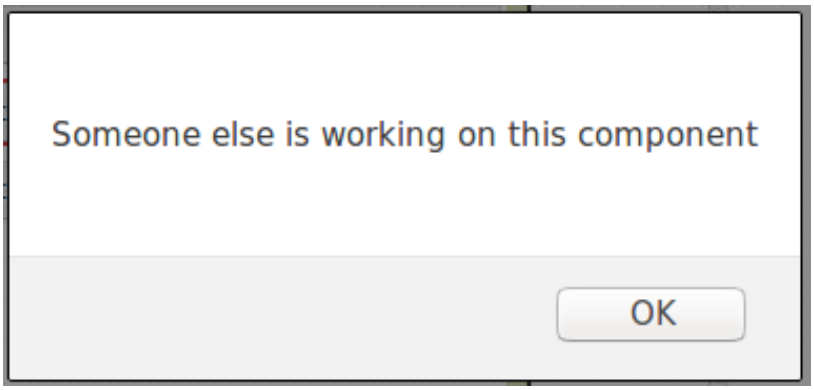


Figure 5-6: Error message when users modify components or blocks locked by others

When users modify the components or blocks locked by other collaborators, they get an error message showing that the components or blocks are locked by others. Figure 5-6 shows the error message dialogue.

In order to implement the component-level collaboration model, I added more events to the event system on both *Designer* and *Blocks*. Table 5.1 summaries the properties and functionality of the events added.

Event class	Property	When event is fired

LockComponent	• User ID • Component ID	Lock the component of the given component ID by the user of the given user ID in the *Designer*
UnlockComponent	• User ID • Component ID	Unlock the component of the given component ID by the user of the given user ID in the *Designer*
LockBlock	• User ID • Block ID	Lock the block of the given block ID by the user of the given user ID in the *Blocks*
UnlockBlock	• User ID • Block ID	Unlock the block of the given block ID by the user of the given user ID in the *Block*

Table 5.1: Summary of the properties and functionality of events added in the component-level collaboration model

5.2 User Study

In order to study the efficacy of the collaboration and understand users' behavior with different collaboration models, I designed and ran a user study. Twenty MIT students over 18 years old participated in the study, and they ranged from freshman to graduate students. Participants were not required to have any experience in programming or MIT App Inventor. In order to be familiar with the interface, I sent them a tutorial

task before the study, but some participants did not finish it. They were paired into groups of two, and each group was asked to complete a task in an hour. Six groups were assigned the task A, *Space Invaders*, as shown in the Appendix A, and four groups were assigned the task B, *Get the Gold*, as shown in the Appendix B. Two tasks have the same level of difficulty and they use the same set of components. Before the study, participants need to fill out the pre-study survey as shown in the Appendix C, which asked about their experience in group projects and blocks-based programming. After the study, they need to fill out the post-study survey as shown in the Appendix D, which asked about the advantages and disadvantages of the collaboration models they used. In order to capture the oral communication between users, we video recorded the study. In the collaboration server, we recorded every event sent between users, so that we can analyze the collaborative behavior afterwards.

5.3 Metrics

To quantify the efficacy of different collaboration models, I defined the metrics introduced in the following subsections.

5.3.1 Average and maximum length of turns

In every collaboration model, two programmers edit the projects by turns. With the project-level collaboration model, programmers switch turns explicitly by clicking the "Switch Leader" button, but in the other two collaboration models, programmers switch turns unintentionally. We ranked all operations of a project in chronological order, and the length of a turn is the number of sequential operations performed by the same user. Average length of turns (ALT) is defined as following:

$$ALT = \frac{\text{Sum of the length of each turn}}{\text{the number of turns}}.$$

The maximum length of turns (MLT) is the maximum length of turns between two users. We used ALT and MLT to characterize the collaborative behavior between

users, such as how often users switch turns, and how long each individual controls the project. The lower ALT and MLT mean that users work more often in parallel.

5.3.2 Communication rate

We recorded each group to track their oral communication, and analyzed how they behaved with different collaboration models. To quantify how much they communicate, we counted how long they talked with each other and worked on the same computer, and then divided it by the time of the study to get the communication rate of each group.

5.3.3 Collaboration rate

We considered that collaboration happens when a user works on top of other collaborators' work. There are several cases when this happens in MIT App Inventor:

1. One user creates a parent block or component, and the other user adds child blocks or components to it.

2. One user creates a component or a block, and the other user modifies its properties.

3. Two users work on the same component or set of blocks. This usually occurs when users fix errors for others or take over others' work.

A project is represented as a tree, and it has a component root node that represents all components in *Designer*, and a block root node that represents all blocks in *Blocks*. There are three kinds of nodes in the tree:

1. Component node: It represents a component in *Designer* or a block in *Blocks*. Each component node has one parent node, and multiple child nodes, including other component nodes and its property nodes.

2. Property node: It represents a property of a component or a block, and it does not have any child nodes.

3. Operation node: It represents a users' operation on the component or property, such as moving a component and changing the value of a property. Since the workspace is two-dimensional, and users can move blocks or components multiple times to place them correctly, we treat the consecutive operations on the same component from the same user as one operation node. For example, if the user A moves a block twice in a row, and user B moves the block once, there are two operation nodes linked to this block, although three operations occurred. Each component node and property node have a list of operation nodes.

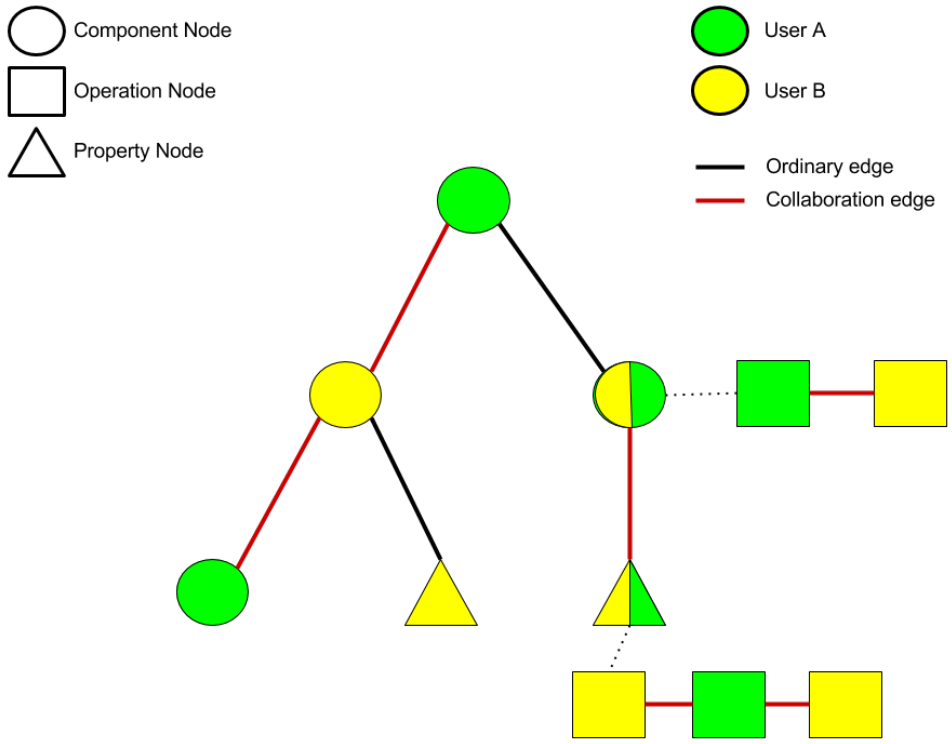


Figure 5-7: An example of a sub-tree of a project illustrating the collaboration edge and different kinds of nodes

We connected component nodes and their children, and the operation nodes of each component node and property node in chronological order with an edge. We colored every node in the tree with the colors of its modifiers, and nodes can have multiple colors if different users work on them. A collaboration edge is defined as the two nodes it connects have different colors, which means users work on top of other collaborators'

work. Figure 5-7 shows an example of a sub-tree of a project, where user A is the green, and user B is yellow. The dotted line connects component node or property node with its list of operation nodes, but it is not an edge.

The collaboration rate (CR) is defined as following:

$$CR = \frac{\text{Number of collaboration edges}}{\text{Total number of edges}}.$$

For example, the sub-tree shown in Figure 5-7 has collaboration rate 0.75. The collaboration rate reflects how much percentage of the project is done by the mixture of two users' work, which depends on how collaborators split their tasks, and the level of their programming skills. For example, if two experienced programmers split tasks wisely, the collaboration rate is expected to be low. Alternatively, if two users are actively working on debugging a problem together, they might have a high collaboration rate.

5.3.4 Mistake rate

Another aspect of efficacy of collaboration is that how many mistakes or unnecessary operations occur during the collaboration. Given the final state of a project tree, we can count the minimal number of operations that were needed to achieve the final state. Therefore, the mistake rate (MR) of project is defined as:

$$MR = \frac{\text{Total Number of operations} - \text{Minimal number of operations}}{\text{Total number of operations}}.$$

It is important to know that users have different experience on MIT App Inventor, so they can aim at wrong implementation of a task. Even though the final state of a project tree can be significantly different from the correct solution, it is interesting to know how users achieve such a state, as long as the group agreed on the implementation. Therefore, MR does not measure the correctness of the program, instead it measures how often users correct themselves. The MR can be higher when the users try different components for a task, which introduces more unnecessary operations to

the project.

5.4 Result

We labeled ten groups from G1, G2 to G10, and Table 5.2 summaries the assignment of groups on the tasks and collaboration models.

	Project-level	Component-level	Real-time
Task A	G1, G4	G2, G5	G3, G6
Task B	G7	G8	G9, G10

Table 5.2: Assignment of groups on tasks and collaboration models

Table 5.3 shows the experience in programming and blocks-based programming tools of each group based on the pre-study survey.

Group	Experience
G1	Both participants have knowledge in programming, but none of them have experience in blocks-based programming tools.
G2	Both have knowledge in programming, but none of them have experience in blocks-based programming tools.
G3	One participant is very familiar with MIT App Inventor, and the other has no experience in blocks-based programming tools.
G4	Both participants have little knowledge in programming, and no experience in blocks-based programming tools.
G5	One participant did the App Inventor tutorial, and the other has no experience in blocks-based programming tools.

G6	One participant has experience in Scratch, and did the MIT App Inventor tutorial. The other participant have knowledge in programming, but no experience in blocks-based programming tools.
G7	One participant has knowledge in Scratch, and the other participant has experience in Lego Mindstorms.
G8	Both participants have knowledge in programming, and one participant has knowledge in Scratch.
G9	One participant has experience in Scratch, and the other participant has experience in Scratch and MIT App Inventor.
G10	Both participants have little knowledge in programming, and no experience in blocks-based programming tools.

Table 5.3: Experience of each group

The user study lasted about one hour, but students were welcomed to stay longer. Since the group was not required to finish the task, we counted how much percentage of task they completed, and calculated the adjusted time they needed to finish the task. Table 5.4 summarizes the adjusted time of each group based on different collaboration models.

Collaboration Model	Group	Actual Time	Completion Percentage	Adjusted Time
Project-level	G1	30	35%	85.71
	G4	45	40%	112.5
	G7	85	90%	94.44
Component-level	G2	55	80%	68.75
	G5	42	55%	76.36
	G8	53	85%	62.35

Real-time	G3	41	100%	41
	G6	49	65%	75.38
	G9	65	90%	72.22
	G10	40	65%	61.54

Table 5.4: Adjusted completion time of each group, time is in minutes.

Since participants had various background on MIT App Inventor, and the study focused on how participants collaborate instead of technical skills, we allowed them to ask for help. It is hard to quantify how the help affects the completion time, but the data suggests that it takes longer to finish the tasks using project-level collaboration model than the other two collaboration models, which is consistent with our expectation, because only one user can edit the project using project-level collaboration model.

5.4.1 Average and maximum length of turns

Table 5.5 and Table 5.6 show the average and maximum length of turns of each group based on different collaboration models. We noticed that ALT and MLT decrease in

Group	Project-level	Group	Component-level	Group	Real-time
G1	68	G2	6.32	G3	4.92
G4	8.36	G5	6.80	G6	3.98
G7	42.21	G8	4.59	G9	4.33
				G10	4.18

Table 5.5: Average length of turns based on different collaboration models

Group	Project-level	Group	Component-level	Group	Real-time
G1	124	G2	46	G3	39
G4	29	G5	49	G6	21
G7	220	G8	65	G9	26
				G10	29

Table 5.6: Maximum length of turns based on different collaboration models

the order of project-level, component-level, and real-time collaboration models. This

is reasonable because in project-level collaboration model, the cost of switching leader is larger compared to the other collaboration models. One exception in project-level collaboration model is G2. We found that the users in G2 switched leader often without informing their partner and they mostly focused on their own computers instead of working on the same computer. The other two groups using project-level collaboration model, G4 and G7, worked on the same computer mostly.

5.4.2 Communication rate

Group	Project-level	Group	Component-level	Group	Real-time
G1	78.61%	G2	38.64%	G3	52.24%
G4	48.15%	G5	48.21%	G6	42.52%
G7	70.3%	G8	37.11%	G9	60.38%
				G10	62.71%
Average	65.69%		41.32%		54.46%

Table 5.7: Communication rate based on different collaboration models

Table 5.7 shows the communication rate of each group based on different collaboration models. The groups using project-level collaboration model have the highest communication rate because they focused on one computer most of time. One typical collaboration strategy they used is that one user edited the project in the MIT App Inventor, and the other opened the task description and figured out what they needed to do next. G2 is an exception, because participants kept working on their own computer and lacked communication, which also resulted in longer adjusted completion time as shown in Table 5.4.

The groups using component-level collaboration model have a lower communication rate than those using real-time collaboration model. By analyzing the video, we found that with component-level collaboration model, participants split tasks first based on components and blocks, and then they tried to solve the task on their own without communicating. Instead, using real-time collaboration model, some groups

did not split tasks explicitly and they discussed solutions together first and then worked on the project. In addition, we noticed that participants in G5 have bigger difference between the experience in MIT App Inventor than those in G2 and G8. Therefore, one user in G5 spent more time on explaining tasks and solutions to the other, which resulted in higher communication rate. Within groups using real-time collaboration model, G3 and G6 have lower communication rates. One participant in G3 is very familiar to MIT App Inventor platform, so he did most of the work in the project, and taught the other how to complete the task. They did not spend much time on discussing the solutions, instead most of communication occurred when debugging. For G6, there were server problem on synchronization when they did the study, so the communication rate is lower compared to other groups.

5.4.3 Collaboration rate

Group	Project-level	Group	Component-level	Group	Real-time
G1	25.85%	G2	34.81%	G3	16.19%
G4	41.91%	G5	23.63%	G6	24.18%
G7	25.61%	G8	25.51%	G9	35.00%
				G10	41.60%
Average	31.12%		27.98%		29.24%

Table 5.8: Collaboration rate based on different collaboration models

Table 5.8 shows the collaboration rate based on different collaboration models. The collaboration rate is affected by how each group collaborated. G4 switched leader more often and communicated less compared to other two groups using project-level collaboration model, so each member worked on top of others' work more often. Within the groups using component-level collaboration model, we noticed that G2 and G8 had a similar communication rate, but their collaboration rate is different. By analyzing their videos, we found that they adopted different collaboration strategy. Both groups split the tasks first and worked on their own computers, but when debugging, G2 used one computer, while G8 fixed bugs on separate computer related to their own tasks. Another important factor is the users' experience. Since one

participant of G3 is very experienced in MIT App Inventor, he did not need the other to fix his work, which results in a lower collaboration rate. In addition, I found that using real-time collaboration models, participants moved others' blocks more often than using other two collaboration models. One reason is that they moved others' blocks to make space for new blocks they worked on, and another reason is that task arrangement using real-time collaboration is not explicitly, so they switched tasks more often.

The average collaboration rate does not vary significantly across different collaboration models because our sample size is too small to mitigate the effect of other factors, such as experience and collaboration strtegy.

5.4.4 Mistake rate

Group	Project-level	Group	Component-level	Group	Real-time
G1	68.14%	G2	61.25%	G3	53.43%
G4	67.52%	G5	63.96%	G6	75.74%
G7	68.36%	G8	80.86%	G9	70.88%
				G10	65.59%
Average	68.01%		68.69%		66.41%

Table 5.9: Mistake rate based on different collaboration models

Table 5.9 summarizes the mistake rate of each group based on different collaboration models. After analyzing the data, I think although mistake rate reflects how much participants correct themselves, but it does not show how well participants did regarding to the correct solution of the project. Since we did not require each group to complete the task, the final state of their project could be significantly different from the correct solution. For example, G4 had similar mistake rate compared to G1 and G7, but its adjusted completion time is much longer. However, mistake rate shows that experience is an important factor. G3 has the lowest mistake rate, since one participant was familiar with the platform and made less mistakes. In contrast, participants in G8 added many unnecessary components at the first, and resolved the issue by requesting help.

5.4.5 Likeability

In addition to the above metrics, we asked participant to answer three same likert scale problems before and after the study where 1 is "Strongly disagree" and 5 is "Strongly Agree". To evaluate how different collaboration models improve their likeability in programming with others, we counted the percentage of participants whose likeability towards each statement increases, remains the same or decreases after finishing the study. Here are the three likert scale statements we asked in the survey, and Table 5.10, 5.11, 5.12 summarizes likeability changes towards each statement based on different collaboration models respectively.

1. I like to work with others when I'm programming.

	Project-level	Component level	Real-time
Increase	16.67%	16.67%	37.5%
Same	50%	66.67%	62.5%
Decrease	33.33%	16.67%	0%

Table 5.10: Likeability change towards statement "I like to work with others when I'm programming." based on different collaboration models

2. Programming with others is helpful for solving problems.

	Project-level	Component level	Real-time
Increase	0%	16.67%	25%
Same	83.33%	83.33%	62.5%
Decrease	16.67%	0%	12.5%

Table 5.11: Likeability change towards statement "Programming with others is helpful for solving problems." based on different collaboration models

3. It is/would be useful to be able to program with others in real-time (on the same code).

Based on the data, we found that users' satisfaction toward collaborative programming is lowest with project-level collaboration model, and highest with real-time collaboration model. In the post-study survey, some participants using project-level

	Project-level	Component level	Real-time
Increase	16.67%	0%	62.5%
Same	50%	83.33%	25%
Decrease	33.33%	16.67%	12.5%

Table 5.12: Likeability change towards statement "It is/would be useful to be able to program with others in real-time (on the same code)." based on different collaboration models

programming mentioned that "Switching leader slows them down" and "It would be great if they can edit the project at the same time".

Chapter 6

Discussion

6.1 Collaboration models

The three collaboration models we proposed illustrate how we control the individual power with different constraints. Gutwin et al. discussed three situations about the trade-off between individual power and group awareness in the collaborative programming environment, and we adapted those inside MIT App Inventor: workspace navigation, that who is allowed to toggle the editors and move around in workspace; artifact manipulation, that who can modify the components and blocks; view representation, that how users are represented in other collaborators' screen [21]. For workspace navigation and view representation, the three collaboration models adopt the same technique. Users are free to toggle the editors and move around the workspace to focus on the part they are interested. For example, one user can open the *Designer* and the other can open the *Blocks*. They are represented as colored squares on other collaborators' screen, and they have a specific color to indicate which part of the project they are working one. The major difference between the three collaboration models is artifact manipulation, since they have the different constraints on editing the project.

Using different collaboration models, participants showed different collaborative behaviors, and the efficiency of collaboration depends on how users take advantage of the constraints. For example, within the groups using project-level collaboration

model, G4 adopted different collaboration strategy from G1 and G7, and it ended up the longest adjusted completion time. It is important that users communicate with each other using project-level collaboration model, and collaboration is less efficient by switching leader too often without enough communication.

Although our sample size is small, I think real-time collaboration model is the most efficient among three collaboration models. The data suggests that it takes shorter time to complete the task using real-time collaboration model than project-level collaboration model, and users were engaged in more communication and collaboration than component-level collaboration model. In addition, the real-time collaboration model improves users' likeability towards collaborative programming most. More participants would be required in order to determine whether those effects are statistically significant. I think the real-time collaboration model is more intuitive for users to understand than the other two collaboration models, since people expect to edit the project at the same time while using collaborative environment such as Google Docs.

6.2 Collaborative programming

The collaborative programming environment I introduced in this thesis is the first attempt that enables users program collaboratively in real time in MIT App Inventor. With the current version of MIT App Inventor, users can only work on a project with one account, which means they cannot edit the project in parallel. Although AIMerger enables users program collaboratively by splitting tasks by screens, it does not allow users to work on the same project and see others' changes simultaneously [13].

The new environment provides users a new approach to teach and learn. For example, it enables "teacher-student" or "mentor-mentee" roles inside MIT App Inventor. Teachers can share the projects with students in read-only mode to demonstrate ideas and demos. Students can work on group projects after school, because they can collaborate remotely. In addition to the commonly used pair programming method, our collaborative programming environment introduced a new mode of cooperation

between students. Instead of sitting shoulder-to-shoulder and working on the same machine, students can work on the different machines in distributed locations and review others' changes simultaneously. Therefore, the new collaborative programming environment will encourage more group projects within the curriculum.

As MIT App Inventor is built for students and novice programmers, the collaborative programming environment gives them an opportunity to develop their teamwork skill at an early stage. Based on the interview with professional software engineers done by Bacchelli et al, they view collaboration from four perspectives: 1. communication, 2. helping each other by sharing information, 3. knowing what others know, and 4. working on the same goal, doing different things [3]. Our real-time collaborative environment addresses three perspectives except the first one. It enables users to work on the same project, and they can work on different components or blocks. Since users can toggle editors and move around workspace freely and the project is identical across different users, they have the same information about the project. Because we represent users with colored square and indicate the part they work on with colored border, users share the information about their actions to others.

Our implementation of collaborative programming environment can be generalized to other blocks-based programming tools, such as Scratch, because we did not use any techniques specific to MIT App Inventor. The publish-subscribe pattern and event handling design pattern can be used in any systems.

6.3 Computational thinking

This new collaboration mechanism for MIT App Inventor touches on all four of the key computational thinking practices of Brennan and Resnick [25]. Multiple users can incrementally and iteratively build small units either in isolation or together depending on the complexity of the tasks and expertise of the individuals. Users can explore different debugging techniques to assist one another in addressing problems in the code. Reuse and remix of code can happen on a much finer time granularity on the order of seconds or minutes. Lastly, users can work together to help one another

understand and exploit abstraction and modularization techniques within a program.

6.4 Limitation

Our collaborative environment does not address the communication perspective and it is a challenge to understand others' thought process when collaborating with blocks-based programming tools. With the text-based programming languages, programmers can know others' plan via comments. However, it is hard to place comments in visual programming environment without disrupting actual programming logic. One way we can handle it is to add a screen for comments, so users can toggle the comments screen as they need. Another way to help users to understand others is adding a communication channel, so that users can exchange their ideas while they are programming.

One problem participants had in the user study is that the project was not synchronized sometime. The failure in the synchronization interrupts the collaboration and decreases users' satisfaction towards collaborative programming. Since we only have one collaboration server, it is easy to have server overloaded and failures when network is not reliable. One solution is that we build a distributed system infrastructure in the collaboration server, such as Paxos or Raft, to achieve consensus across different clients. Another approach is using web real-time communication toolkit, such as WebRTC [24], to establish peer-to-peer connection between clients, so that client can not only request data from the server, but also from browsers of other clients.

Chapter 7

Conclusion

In this thesis, I present a real-time collaborative programming environment inside MIT App Inventor, which enables users work on the same project, and see others' changes simultaneously. I designed and modified the collaborative environment to support three collaboration models: 1. Project-level collaboration model, 2. Component-level collaboration model, and 3. Real-time collaboration model. I conducted a user study with twenty participants to evaluate the efficacy of different collaboration models, and concluded that using real-time collaboration model, users have more control over the project, they finish the tasks faster and engage in more communication, and they have a higher likeability towards collaborative programming than using other two collaboration models. When the real-time collaborative programming environment is published to the public, it will change the design of the curriculum based on MIT App Inventor, and more complex and interesting applications will appear.

7.1 Future work

To publish the collaborative programming environment, there are a few improvements we can make to the system.

1. To enable user communicate when collaborating in distributed locations, we can build a chat window inside MIT App Inventor. The chat window allows users to communicate via text, audio or video.

2. To improve the scalability and synchronization problem, we can deploy WebRTC to enable users request data from the server and browsers of other clients, so that we mitigate the server overload and the lag of synchronization.

Appendix A

User Study Task A : Space Invaders

The Space Invaders App features a player controlled rocket and an alien spaceship.The player scores points by hitting the alien spaceship with a bullet while the alien spaceship appears in random locations.

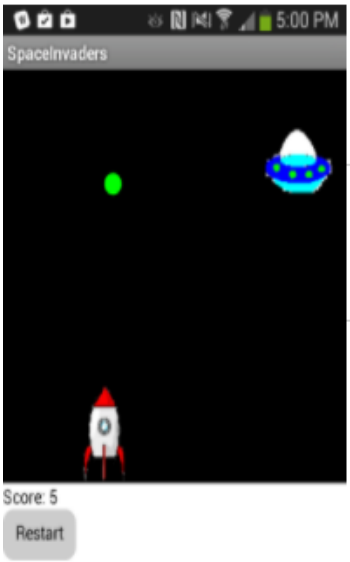


Figure A-1: The user interface of the Space Invaders App

Components you would use:

- Clock
- ImageSprite
- Canvas
- Label
- Button

App Functionality:

1. Initialize Game:

   Score should be zero, and bullet should be invisible, put rocket and saucer at an initial position.

2. Restart Button:

   Sometimes, users might want to restart the game and reset their score. Users can do this by clicking the Reset Button. When this happens, we need to set the score back to 0, and reset rocket and saucer's position.

3. Score Label:

   This label shows the current score.

4. Game Play:

   User can drag the rocket only along its X axis. When the rocket is dragged set its X property to be the currentX that we dragged the sprite to.

   When user touches the rocket, we fired a bullet whose heading is 90 and speed is 5. Bullet is a Ball component on the canvas. Heading is a value from 0 to 360 that indicates what direction the sprite should be moving towards. 0/360 is to the left, 90 is up, 180 is right, and 270 is down. The speed is measured in pixels/sec.

   If the bullet hit the saucer, the score should increase by 1. Bullet should become invisible, and the saucer should move to a new position. If the bullet hits the top edge of our canvas, it should disappear.

   To make the game even harder, we set a Clock component, which has 3000 time interval. When the clock goes off, we reset saucer's position.

Appendix B

User Study Task B : Get the Gold

Get the Gold app features a player controls a pirate ship to collect gold. The goal of the game is to collect all coins on the screen. Components you would use:

Figure B-1: The user interface of the Space Invaders App

- Button
- Label
- ImageSprite
- Canvas
- Clock

App Functionality:

1. Game Initialization:

   There are five coins on the screen, and a pirate ship. When game starts, place coins and ship at random positions on the canvas.

2. Time Label:

   Create a label to display the time that it took you to get all the gold. It's initial state is 0:0.

3. Reset Button:

   When reset button is clicked, reset game to its initial state.

4. Clock:

   You will need a clock component. Set its time interval to 2000. When timer goes off, move the coins that have not been collected by the ship to new random positions.

5. Pirate Ship: Set the initial speed to 6. When ship is flung, set the heading of the ship to the new heading. When ship hit the edge, bounce ship back. When ship collides with the coins, make the coin disappear on the screen

Appendix C

App Inventor Group Collaboration Pre-study User Survey

1. What year are you?
    - Freshman
    - Sophomore
    - Junior
    - Senior
    - Graduate Student
2. Which major are you?
3. Have you ever participated in group projects, including class assignment? If yes, what is the largest team size?
4. Have you used any of the following block-based programming tools?
    - App Inventor
    - Snap!
    - Scratch
    - Lego Mindstorms

5. Have you ever built an Android application?

6. On a scale of 1-5 (with 1 being the lowest and 5 being the highest) please rate the following statements

    - I like to work with others when I'm programming.
    - Programming with others is helpful for solving problems.
    - It is/would be useful to be able to program with others in real-time (on the same code).

Appendix D

App Inventor Group Collaboration Post-study User Survey

1. Please circle the collaborative mode you did with and write down the task name you did:
    - Project-level collaborative mode:
    - Component-level collaborative mode:
    - Real-time collaborative mode:
2. In this collaborative mode, please explain in detail the ways used to identify what your teammate was doing within the App Inventor interface.
3. Please describe any instances during this collaborative mode that you found it particularly useful to collaborate with your partner in real time.
4. Please describe any instances during this collaborative mode that you found it particularly frustrating to collaborate with your partner in real time.
5. What changes would you make to the system if you are the engineer in App Inventor?
6. Do you have any additional comments about this collaborative modes, or the real-time collaboration in App Inventor in general?

7. On a scale of 1-5 (with 1 being the lowest and 5 being the highest) please rate the following statements

    - I like to work with others when I'm programming.
    - Programming with others is helpful for solving problems.
    - It is/would be useful to be able to program with others in real-time (on the same code).

Bibliography

[1] "About Us." *About Us | Explore MIT App Inventor*. MIT App Inventor Team, n.d. Web. 19 Apr. 2017.

[2] D. Spinellis, "Version control systems," in IEEE Software, vol. 22, no. 5, pp. 108-109, Sept.-Oct. 2005.

[3] Guzzi, Anja, Alberto Bacchelli, Yann Riche, and Arie Van Deursen. "Supporting Developers'Coordination in the IDE." Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15 (2015)

[4] Boyer, Kristy Elizabeth, August A. Dwight, R. Taylor Fondren, Mladen A. Vouk, and James C. Lester. "A Development Environment for Distributed Synchronous Collaborative Programming." ACM SIGCSE Bulletin 40.3 (2008): 158.

[5] Goldman, Max, Greg Little, and Robert C. Miller. "Real-time Collaborative Coding in a Web IDE." Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology - UIST '11 (2011)

[6] Ghorashi, Soroush, and Carlos Jensen. "Jimbo." Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering - CHASE '16 (2016)

[7] Suzuki, Hideyuki, and Hiroshi Kato. "AlgoBlock âĂŤan Open Programming Language." Interaction-level Support for Collaborative Learning. L. Erlbaum Associates Inc., Oct. 1995

[8] McKinsey, Jonathan. "Remote Pair Programming in a Visual Programming Language." Thesis. EECS Department, University of California, Berkeley, 2015.

[9] Maloney, John, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. "The Scratch Programming Language and Environment." ACM Transactions on Computing Education 10.4 (2010)

[10] Wolber, David, Hal Abelson, Ellen Spertus, and Liz Looney. App Inventor 2: Create Your Own Android Apps. Beijing: O'Reilly, 2014

[11] "Overview." GWT Project. N.p., n.d. Web. 01 May 2017. <http://www.gwtproject.org/overview.html>.

[12] "Blockly | Google Developers." Google. Google, n.d. Web. 01 May 2017. <https://developers.google.com/blockly/>.

[13] Feeney, Katherine Kyle. "ENCOURAGING COLLABORATION THROUGH APP INVENTOR." Thesis. Mills College, 2012. Web.

[14] Gross, Tom. "Supporting Effortless Coordination: 25 Years of Awareness Research." Computer Supported Cooperative Work (CSCW) 22.4-6 (2013): 425-74.

[15] Dourish, Paul, and Sara Bly. "Portholes." Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '92 (1992): n. pag.

[16] Google Docs. Google Inc., n.d. Web. <https://www.google.com/docs/about/>.

[17] Gamma, Erich, Richard Helm, Ralph E. Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. New Dehli: Pearson Education, 2015. Print.

[18] Socket.IO. Web. 05 May 2017. <https://socket.io/>.

[19] "Pub/Sub." Redis. Web. 05 May 2017. <https://redis.io/topics/pubsub>.

[20] Bender, Matthias, Sebastian Michel, Sebastian Parkitny, and Gerhard Weikum. "A Comparative Study of Pub/Sub Methods in Structured P2P Networks."

Databases, Information Systems, and Peer-to-Peer Computing Lecture Notes in Computer Science (n.d.): 385-96.

[21] Gutwin, Carl, and Saul Greenberg. "Design for Individuals, Design for Groups." Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work - CSCW '98 (1998): n. pag. Web.

[22] Cockburn, Alistair, and Laurie Williams. "The costs and benefits of pair programming." Extreme programming examined (2000): 223-247.

[23] McDowell, Charlie, et al. "Pair programming improves student retention, confidence, and program quality." Communications of the ACM 49.8 (2006): 90-95.

[24] "WebRTC Home | WebRTC." WebRTC Home | WebRTC. N.p., n.d. Web. 19 May 2017. <https://webrtc.org/>.

[25] Brennan, Karen, and Mitchel Resnick. "New Frameworks for Studying and Assessing the Development of Computational Thinking." Proceedings of the 2012 Annual Meeting of the American Educational Research Association, Vancouver, Canada (2012): n. pag. Web.