CHAPTER 14

Understanding an App's Architecture

This chapter examines the structure of an app from a programmer's perspective. It begins with the traditional analogy that an app is like a recipe and then proceeds to reconceptualize an app as a set of components that respond to events. The chapter also examines how apps can ask questions, repeat, remember, and talk to the Web, all of which will be described in more detail in later chapters.

Many people can tell you what an app is from a user's perspective, but understanding what it is from a programmer's perspective is more complicated. Apps have an internal structure that we must fully understand in order to create them effectively.

One way to describe an app's internals is to break it into two parts, its *components* and its *behaviors*. Roughly, these correspond to the two main windows you use in App Inventor: you use the Component Designer to specify the objects (components) of the app, and you use the Blocks Editor to program how the app responds to user and external events (the app's behavior).

Figure 14-1 provides an overview of this app architecture. In this chapter, we'll explore this architecture in detail.

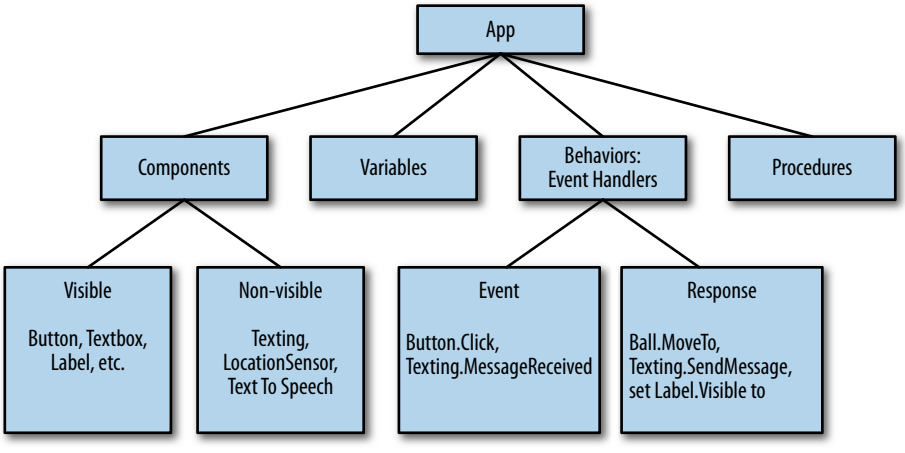

Figure 14-1. The internal architecture of an App Inventor app

Components

There are two main types of components in an app: visible and non-visible. The app's visible components are the ones you can see when the app is launched—things like buttons, text boxes, and labels. These are often referred to as the app's *user interface*.

Non-visible components are those you can't see, so they're not part of the user interface. Instead, they provide access to the built-in functionality of the device; for example, the `Texting` component sends and processes SMS texts, the `LocationSensor` component determines the device's location, and the `TextToSpeech` component talks. The non-visible components are the technology within the device—little people that do jobs for your app.

Both visible and non-visible components are defined by a set of *properties*. Properties are memory slots for storing information about the component. Visible components, for instance, have properties like `Width`, `Height`, and `Alignment`, which together define how the component looks. So a button that looks like the Submit button in Figure 14-2 to the end user is defined in the Component Designer with a set of properties, including those shown in Table 14-1.

Table 14-1. Button properties

Width	Height	Alignment	Text
50	30	center	Submit

Figure 14-2. Submit button

You can think of properties as something like the cells you see in a spreadsheet. You modify them in the Component Designer to define the *initial* appearance of a component. If you change the number in the `Width` slot from 50 to 70, the button will appear wider, both in the Designer and in the app. Note that the end user of the app doesn't see the 70; he just sees the button's width change.

Behavior

An app's components are generally straightforward to understand: a text box is for entering information, a button is for clicking, and so on. An app's behavior, on the other hand, is conceptually difficult and often complex. The behavior defines how the app should respond to events, both user initiated (e.g., a button click) and external (e.g., an SMS text arriving to the phone). The difficulty of specifying such interactive behavior is why programming is so challenging.

Fortunately, App Inventor provides a visual "blocks" language perfectly suited for specifying behaviors. This section provides a model for understanding it.

An App As a Recipe

Traditionally, software has often been compared to a recipe. Like a recipe, a traditional app follows a linear sequence of instructions, such as those shown in Figure 14-3, that the computer (chef) should perform.

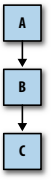

Figure 14-3. Traditional software follows a linear sequence of instructions

A typical app might start a bank transaction (A), perform some computations and modify a customer's account (B), and then print out the new balance on the screen (C).

An App As a Set of Event Handlers

However, most apps today, whether they're for mobile phones, the Web, or desktop computers, don't fit the *recipe* paradigm anymore. They don't perform a bunch of instructions in a predetermined order; instead, they *react to events*—most commonly, events initiated by the app's end user. For example, if the user clicks a button, the app responds by performing some operation (e.g., sending a text message). For touchscreen phones and devices, the act of dragging your finger across the screen is another event. The app might respond to that event by drawing a line from the point of your original touch to the point where you lifted your finger.

These types of apps are better conceptualized as a set of components that respond to events. The apps do include "recipes"—sequences of instructions—but each recipe is only performed in response to some event, as shown in Figure 14-4.

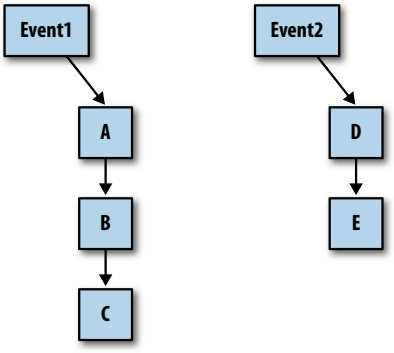

Figure 14-4. An app as multiple recipes hooked to events

So, as events occur, the app reacts by calling a sequence of *functions*. Functions are things you can do to or with a component—operations like sending an SMS text, or property-changing operations such as changing the text in a label of the user interface. To *call* a function means to *invoke* it, to make it happen. We call an event and the set of functions performed in response to it an *event handler*.

Many events are initiated by the end user, but some are not. An app can react to events that happen within the phone, such as changes to its orientation sensor and the clock (i.e., the passing of time), and events created by things outside the phone, such as other phones or data arriving from the Web, as shown in Figure 14-5.

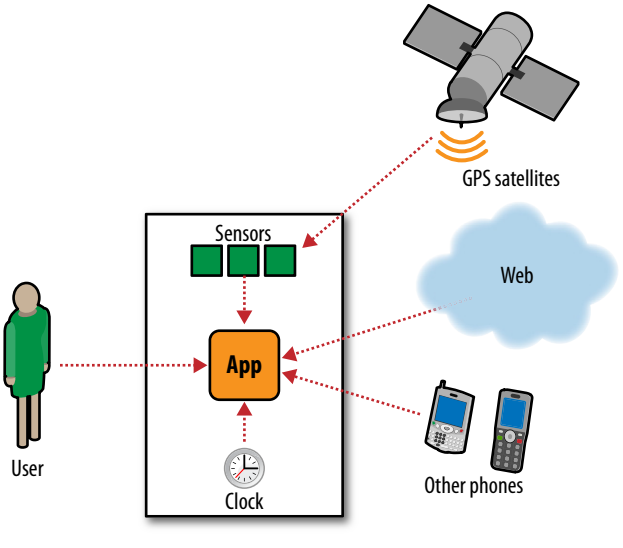

Figure 14-5. An app can respond to both internal and external events

One reason App Inventor programming is intuitive is that it's based directly on this event-response paradigm; event handlers are primary "words" in the language (in many languages, this is not the case). You begin defining a behavior by dragging out an *event block*, which has the form, "When <event> do". For example, consider an app, SpeakIt, that responds to button clicks by speaking the text the user has entered aloud. This application could be programmed with a single event handler, shown in Figure 14-6.

Figure 14-6. An event handler for a SpeakIt app

These blocks specify that when the user clicks the button named `SpeakItButton`, the `TextToSpeech` component should speak the words the user has entered in the text box named `TextBox1`. The response is the call to the function **TextToSpeech1.Speak**. The event is **SpeakItButton.Click**. The event handler includes all the blocks in Figure 14-6.

With App Inventor, all activity occurs in response to an event. Your app shouldn't contain blocks outside of an event's "when-do" block. For instance, the blocks in Figure 14-7 don't make sense floating alone.

Figure 14-7. Floating blocks won't do anything outside an event handler

Event Types

The events that can trigger activity fall into the categories listed in Table 14-2.

Table 14-2. Events that can trigger activity

Event type	Example
User-initiated event	*When* the user clicks `button1`, *do*…
Initialization event	*When* the app launches, *do*…
Timer event	*When* 20 milliseconds passes, *do*…
Animation event	*When* two objects collide, *do*…
External event	*When* the phone receives a text, *do*…

User-initiated events

User-initiated events are the most common type of event. With input forms, it is typically the button click event that triggers a response from the app. More graphical apps respond to touches and drags.

Initialization events

Sometimes your app needs to perform certain functions right when the app begins, not in response to any end-user activity or other event. How does this fit into the event-handling paradigm?

Event-handling languages like App Inventor consider the app's launch as an event. If you want specific functions to be performed immediately when the app opens, you drag out a **Screen1.Initialize** event block and place some function call blocks within it.

For instance, in the game MoleMash (Chapter 3), the `MoveMole` procedure is called at the start of the app to randomly place the mole, as shown in Figure 14-8.

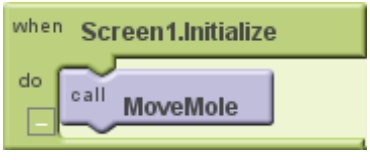

Figure 14-8. Using a Screen1.Initialize event block to move the mole when the app begins

Timer events

Some activity in an app is triggered by the passing of time. You can think of an animation as an object that moves when triggered by a *timer event*. App Inventor has a `Clock` component that can be used to trigger timer events. For instance, if you wanted a ball on the screen to move 10 pixels horizontally at a set time interval, your blocks would look like Figure 14-9.

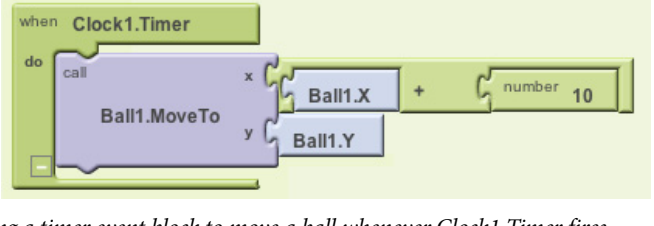

Figure 14-9. Using a timer event block to move a ball whenever Clock1.Timer fires

Animation events

Activity involving graphical objects (sprites) within canvases will trigger events. So you can program games and other interactive animations by specifying what should occur when two objects collide or when an object reaches the edge of the canvas. For more information, see Chapter 17.

External events

When your phone receives location information from GPS satellites, an event is triggered. Likewise, when your phone receives a text, an event is triggered (Figure 14-10).

Such external inputs to the device are considered events, just like the user clicking a button.

So every app you create will be a set of event handlers: one to initialize things, some to respond to the end user's input, some triggered by time, and some triggered by external events. Your job is to conceptualize your app in this way and then design the response to each event handler.

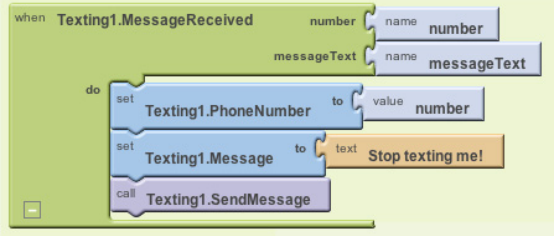

Figure 14-10. The Texting1.MessageReceived event is triggered whenever a text is received

Event Handlers Can Ask Questions

The responses to events are not always linear recipes; they can ask questions and repeat operations. "Asking questions" means to query the data the app has stored and determine its course (branch) based on the answers. We say that such apps have *conditional branches*, as illustrated in Figure 14-11.

In the diagram, when the event occurs, the app performs operation A and then checks a condition. Function B1 is performed if the condition is true. If the condition is false, the app instead performs B2. In either case, the app continues on to perform function C.

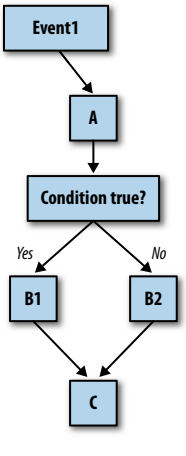

Figure 14-11. An event handler can branch based on the answer to a conditional question

Conditional tests are questions such as "Has the score reached 100?" or "Did the text I just received come from Joe?" Tests can also be more complex formulas including multiple relational operators (less than, greater than, equal to) and logical operators (and, or, not).

You specify conditional behaviors in App Inventor with the **if** and **ifelse** blocks. For instance, the block in Figure 14-12 would report "You Win!" if the player scored 100 points.

Conditional blocks are discussed in detail in Chapter 18.

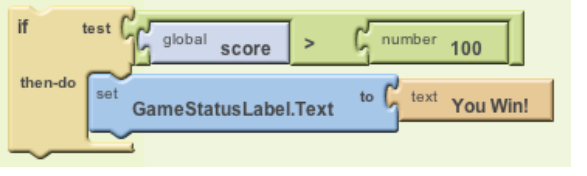

Figure 14-12. Using an if block to report a win once the player reaches 100 points

Event Handlers Can Repeat Blocks

In addition to asking questions and branching based on the answer, your app can also repeat operations multiple times. App Inventor provides two blocks for repeating, the **foreach** and the **while do**. Both enclose other blocks. All the blocks within **foreach** are performed once for each item in a list. For instance, if you wanted to text the same message to a list of phone numbers, you could use the blocks in Figure 14-13.

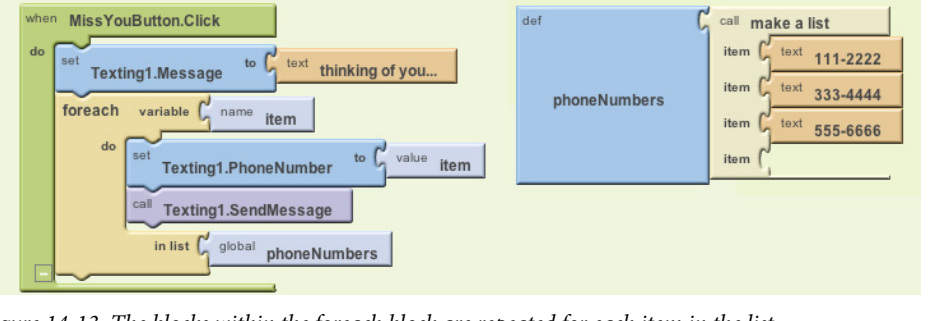

Figure 14-13. The blocks within the foreach block are repeated for each item in the list

The blocks within the **foreach** block are repeated—in this case, three times, because the list **PhoneNumbers** has three items. So the message "thinking of you…" is sent to all three numbers. Repeating blocks are discussed in detail in Chapter 20.

Event Handlers Can Remember Things

Because an event handler executes blocks, it often needs to keep track of information. Information can be stored in memory slots called *variables*, which you define in the Blocks Editor. Variables are like component properties, but they're not associated with any particular component. In a game app, for example, you can define a variable called "score" and your event handlers would modify its value when the user does something accordingly. Variables store data temporarily while an app is running; when you close the app, the data is no longer available.

Sometimes your app needs to remember things not just while it runs, but even when it is closed and then reopened. If you tracked a high score for the history of a game, for example, you'd need to store this data long-term so it is available the next time someone plays the game. Data that is retained even after an app is closed is called *persistent data*, and it's stored in some type of a database.

We'll explore the use of both short-term memory (variables) and long-term memory (database data) in Chapters 16 and 22, respectively.

Event Handlers Can Talk to the Web

Some apps use only the information within the phone or device. But many apps communicate with the Web by sending requests to *web service APIs* (*application programming interfaces*). Such apps are said to be "web-enabled."

Twitter is an example of a web service to which an App Inventor app can talk. You can write apps that request and display your friend's previous tweets and also update your Twitter status. Apps that talk to more than one web service are called *mashups*. We'll explore web-enabled apps in Chapter 24.

Summary

An app creator must view his app both from an end-user perspective and from the inside-out perspective of a programmer. With App Inventor, you design how an app looks and then you design its behavior—the set of event handlers that make an app behave as you want. You build these event handlers by assembling and configuring blocks representing events, functions, conditional branches, repeat loops, web calls, database operations, and more, and then test your work by actually running the app on your phone. After you write a few programs, the mapping between the internal structure of an app and its physical appearance becomes clear. When that happens, you're a programmer!