Ansible: Select-to-Edit for Physical Widgets

Benjamin M. Crowder
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/9266

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.
Ansible: Select-to-Edit for Physical Widgets

Benjamin M. Crowder

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Michael D. Jones, Chair
Kevin Seppi
Ryan Farrell

Department of Computer Science
Brigham Young University

Copyright © 2020 Benjamin M. Crowder
All Rights Reserved
ABSTRACT

Ansible: Select-to-Edit for Physical Widgets

Benjamin M. Crowder
Department of Computer Science, BYU
Master of Science

Ansible brings select-to-edit functionality to physical widgets. When programming sets of physical widgets, it can be bothersome for a programmer to remember the name of the software object that corresponds to a specific widget. Click-to-edit functionality in GUI programming provides a physical action—moving the mouse to a widget and clicking a button on the mouse—to select a virtual widget. In a similar vein, when programming physical widgets, it is natural to point at a widget and think, “I want to program that one.” Ansible allows physical user interface programmers to “click” on a physical widget by making a physical action: shining a light, waving a magnet, or pressing a button on the widget. This brings up the widget’s code for editing on a laptop or workstation. The Ansible system is intended to help physical user interface programmers prototype distributed systems built from physical widgets. We conducted a user study with twelve programmers using Ansible; the study showed that shining a light eliminates the need for a programmer to remember the mapping between physical widgets and their names. We also built three example systems to illustrate some of the kinds of systems that can be implemented using Ansible.

Keywords: physical user interfaces, programming support, development environments
ACKNOWLEDGMENTS

I’m grateful to Heavenly Father for setting me on this journey and helping me so much along the way; to my advisor, Mike Jones, for his advice and help with everything; and to my wife, Meridith, for her perennial support.
# Table of Contents

List of Figures vi

List of Tables vii

1 Introduction 1
  1.1 Ansible ................................................................. 2

2 Related Work 5
  2.1 Programming physical computing systems ............................. 5
  2.2 Widgets for physical computing systems ............................. 9

3 Implementation 10
  3.1 Programming widgets in Ansible .................................... 10
    3.1.1 Widget behaviors .................................................. 11
  3.2 Ansible architecture and protocols .................................. 12
    3.2.1 Hardware ........................................................... 12
    3.2.2 Protocols ........................................................... 14
  3.3 Ansible implementation ................................................. 16
    3.3.1 Software ............................................................ 16
    3.3.2 Widget hardware ................................................... 18

4 User Study 20
  4.1 Methods ............................................................... 20
  4.2 Results ............................................................... 22
4.2.1 Mapping widgets to code ........................................... 22
4.2.2 Confidence in identification ....................................... 23
4.2.3 Frequency of selection .............................................. 24

5 Example Systems .......................................................... 25
5.1 Pill bottle monitor ......................................................... 25
5.2 Room-size adventure game ............................................. 27
5.3 Home door lock checker ............................................... 28

6 Conclusion ................................................................. 31

References ................................................................. 33
List of Figures

1.1 Two variations on select-to-edit functionality for physical widgets as implemented in Ansible. Left: selecting a widget at a distance using a laser pointer. Right: selecting a widget hidden in a stuffed animal using a magnet held in the left hand. In both cases, the programmer does not need to recall the name of a widget to program it. .......................................................... 3

3.1 A programmer uses the laser pointer to select an LED widget and thus insert its name in the button widget’s code. ................................................. 11

3.2 The parts of a widget and the topology of the star network used by Ansible. Programming happens on the server. ................................................. 13

3.3 Inter-widget behavior flow. Only the server maintains a list of connected widgets. 15

3.4 Three of the six implemented Ansible widget types. From left: an LED widget, a button widget, and a proximity sensor widget. Each widget contains an interactive component, a selection sensor, and a Raspberry Pi Zero W computer. 18

5.1 Pill bottle monitor. ................................................................. 25

5.2 Adventure game. ................................................................. 27

5.3 Door lock checker. ................................................................. 29
List of Tables

3.1 Strengths and weaknesses of three widget selection methods implemented in Ansible. ...................................................... 10

4.1 Participants’ use of the light selection modality. ......................... 24
Chapter 1

Introduction

When programming systems from collections of physical widgets, it can be difficult for the programmer to remember which widget is which. This problem comes up when the programmer knows which widget they’d like to reference in a program but cannot remember the name of that widget in the code.

For graphical user interfaces (GUIs), this problem is easily solved in traditional editing tools where the programmer simply double clicks (or right clicks) on the widget in an interface editor to bring up the widget’s code in a window. An early version of click-to-edit functionality appears in Buxton’s MENULAY system [9, figure 6]. More recently, Visual Studio 2017 and earlier versions give programmers the ability to double-click on a widget in an interface preview, opening the main event handler for that widget.

For physical widgets, then, a similar and fairly natural way to solve this problem is to allow the programmer to point at the widgets to program them.

In this paper, we use the term “widget” to refer to physical input devices (buttons, knobs, sensors, etc.), output devices (buzzers, LEDs, screens, etc.), and even smart devices (light bulbs, televisions, etc.). Our use of the term “widget” is intended to emphasize the analogy with GUIs. We also use the term “point” to refer to any of the supported physical motions (pointing, waving, pressing) that selects a widget for programming.

While physical computing systems occur in many contexts, a conversation on a smart home forum illustrates why it is important to solve the problem of widget selection. One person asked [1]: “What do you do when coding in SmartApps or Webcore? How do you keep
[a smart things naming scheme] straight mentally while writing them?” The beleaguered response: “With somewhere between 150 and 200 device names to remember I cannot, and do not want to, learn custom names for each device in the house.”

Merely using a good naming scheme—such as room name - device type—breaks down when there are many widgets, when many widgets look the same, when widgets are physically moved around, or when the programmer needs to modify the system many months or years after writing the code. A blog post describing the setup process for a box of new smart light bulbs illustrates one of the problems with naming schemes: “you won’t necessarily know which one is which. Go ahead and guess for now—if it’s wrong, you can go back into the settings and change it” [18].

Knowing which widget is which is important to programmers, particularly in an object-oriented, event-driven paradigm where each physical widget has a corresponding virtual object representation with event handlers.

1.1 Ansible

We present Ansible, a system which allows programmers to point at a physical widget to select it. The programmer can “point at” a widget by shining a laser at the widget, moving a magnet near the widget, or pressing a button on the widget. Pointing a laser at a widget enables selection when the widget is out of reach, which might happen when the widget is on a high ceiling or across the room. Moving a magnet near a widget requires close proximity but works when the widget is not visible. Pressing a button is simple and does not require having a light or magnet on hand. Two of these methods are shown in Figure 1.1.

These selection mechanisms are implemented in Ansible, an object-oriented, event-driven user interface tool for programming interactive systems composed of physical widgets. Ansible is designed to mimic the functionality of GUI creation tools like MENULAY but in the context of physical widgets.
Figure 1.1: Two variations on select-to-edit functionality for physical widgets as implemented in Ansible. Left: selecting a widget at a distance using a laser pointer. Right: selecting a widget hidden in a stuffed animal using a magnet held in the left hand. In both cases, the programmer does not need to recall the name of a widget to program it.

Selecting a widget either brings up its code (on a laptop or workstation) or inserts its name in the code for another widget. Ansible also helps with the mapping from code to physical widget: selecting the name of a widget in the GUI code editor causes an LED on the widget to flash, to help the programmer remember which widget is which.

We evaluated the usability of Ansible with a qualitative, mixed methods user study in which twelve programmers were tasked with building a small game involving seven widgets. The control group of six programmers used a version of Ansible with traditional click-to-edit functionality, where they had to click on buttons in a GUI to select the widgets. The experimental group of six programmers used Ansible with the light selection modality. Several of the control group participants commented on their experience with creating a mapping between devices and names. Along those lines, participant C6 wrote: “Once I figured out what mapped to what the code was easy to write.” In contrast, none of the participants in the experimental group mentioned mappings; instead, they tended to talk about how easy it was. As E2 said: “using the light to select the widgets was very easy and very nice.”
To demonstrate the kinds of systems that can be built using Ansible, we created three example systems: a pill bottle monitor, an adventure game, and a home door lock checker. Each of these systems involves several widgets and required less than thirty minutes to program.

In the following sections, we discuss related prior work; describe Ansible in increasing levels of detail, starting with the programmer experience and finishing with implementation details; document the user study; detail the three example systems; and close with concluding thoughts.
Chapter 2

Related Work

Ansible is related to and builds on prior work in programming physical computing systems and physical widgets, and in object-oriented, event-driven widget representations.

2.1 Programming physical computing systems

Rädle et al. [20] discuss the problem of using names—specifically in menus, but the principle holds—when referring to spatially organized physical devices: “However, Hamilton and Wigdors own experimentation and user study revealed that keeping track of multiple, often very similar devices can represent a surprisingly significant challenge, and also that users extensively used spatial configurations of tablets for categorical organization. Therefore, it is possible that a non-spatial (or spatially-agnostic) interaction might diminish the benefits of cross-device interaction: It could create a mismatch between the spatial referential domain in which the user’s intention is expressed and the non-spatial way in which the interaction technique requires the user to select a destination device. In other words, although menus are a robust and familiar way to select items that have no clear spatial relation, they might seem cumbersome for the purpose of selecting one out of many devices from a spatial configuration.” Acknowledging that keeping track can indeed be a significant challenge, Ansible presents a spatial interaction that preserves and takes advantage of the user’s spatial organization of the devices, obviating the need for lists of widget names.

Greenberg and Fitchett’s Phidgets (a portmanteau of “physical widgets”) are intended to be “easy enough for the average programmer to extend” [10]. In code, each phidget is
represented by a software object with methods and fields. Each software object class is
instantiated for phidgets connected to the system. The programmer implements an interactive
system by writing event handlers for the phidgets in the system. A central server manages
device connections. Ansible shares the same goal of making physical computing “easy enough”
for the average programmer by hiding electronic and network communication details, and it
uses a similar object-oriented, event-driven programming framework (but in Python rather
than Visual Basic). The .NET Gadgeteer system [28] also includes a similar object-oriented,
event-driven approach to programming physical computing systems, as does the Calder toolkit
[17].

While Ansible uses a similar approach, its widgets can be selected through physical
motions such as pointing a light, waving a magnet, or pressing a button. One conceptual
difference, too, is that an Ansible widget contains a computer as part of the widget, rather
than being connected to a computer. The additional computation can be used to process
data locally—including widget selection events—but also means the widget itself draws more
power, requiring either a large-capacity battery or a connection to a wall outlet.

The Phidgets programming environment includes an interface editor in which the
programmer can drag and drop graphical representations of different phidgets to create and
simulate a system in software before plugging in corresponding hardware phidgets [10, figure
6]. To edit the code for a phidget, the programmer clicks on an ActiveX component on the
screen. Ansible does not do this but instead has the programmer “click” on a physical widget
in the real world (via light, magnet, or button) to select it for programming.

Physical widgets are potentially more useful when they can be physically distributed
rather than needing to be located in close, connected proximity. This idea is seen in Shared
Phidgets [19] and iStuff [5], where self-contained programmable objects with event-driven
interfaces can be distributed over some area. Likewise, Ansible widgets can be distributed
at larger scales—throughout a room, for example. Following Shared Phidgets’ lead, we also
hide network details from the programmer.
Similar to Phidgets, VoodooFlash [25] has a component-based programming environment, but it tightly integrates the physical interface layout—on a prototyping board—with a graphical representation on a connected workstation screen. Widgets are selected in the graphical interface, and manipulating a widget in the real world alters the values in the graphical view of the system. This approach—graphical feedback based on physical manipulation—may simplify problems associated with identifying widgets during programming, but Ansible eschews a graphical representation, choosing to focus on the real-world, physical widgets.

Another way to program physical systems like these is programming by demonstration (PBD). In a PBD system, the programmer performs an action with a sensor, such as pressing a force sensitive resistor. They then use a tool like Exemplar [12] to mark the sensor data at the time of the action and train the system to recognize the action, often with machine learning. The demonstrative element here is echoed in Ansible’s selection modalities, with the difference that in PBD systems, demonstrative interaction is used to capture widget behavior (the trained actions), whereas in Ansible the demonstrative interaction is used to select the widget for programming in the first place. The two techniques can of course live in harmony; while Lau [16] points out some valid flaws of programming by demonstration, PBD holds promise as an ancillary technique. For example, Ansible’s imperative widget API works well for a number of clear-cut cases, but for more complex patterns (especially with analog sensors like the force sensitive resistor), PBD seems a more effective way to capture the expected behavior.

The d.tools system [11] takes a different approach to programming physical systems. Rather than writing event handlers for widgets-as-objects, in d.tools the programmer creates an interactive system in a graphical authoring environment by linking states with transitions. (IFTTTT [2] is somewhat similar in that regard.) A state node represents the combined state of several hardware components. Transitions between state nodes define the rules for moving between states. Rather than working with single widgets with control spread over a distributed system (as is done in Ansible), the d.tools approach allows the programmer
to define states which may combine the state of many individual widgets. Working with these composite states may simplify the problem of identifying widgets in a system of tens or hundreds of widgets. Even so, being able to select an individual widget by pointing at it may nevertheless simplify the process of gathering widget states into a state node.

Ease of use is a key factor in the idea behind Ansible. Bloctopus [21] uses a “friendly” graphical programming environment in which a novice programmer connects nodes—representing hardware elements—using arcs that connect hardware outputs to hardware inputs. A second kind of node can contain JavaScript code for implementing more complex control flow. Intuino [29], MaggLite [13], and one layer of ESPrant SDLK [26] also provide node-based visual languages that allow programmers—usually non-technical users—to construct programs by dragging blocks and drawing connection lines. And LittleBits has hardware modules which are pre-programmed with reasonable behavior [7]. Rather than catering to novice, inexperienced programmers, Ansible is at this point intended more for programmers with somewhat more experience; its ease of use is found in the selection of widgets and to a degree in the widget API, not necessarily in the programming language itself (Python). Similarly to this prior work, however, Ansible does hide electronics details from the programmer, allowing them to focus more on their actual application. Additional investigation would be needed to determine if the select-to-edit functionality of Ansible is relevant for novice programmers.

At the other extreme, some programming environments for physical computing target programmers with specialized skills or interest in electronics and real-time operating systems. These environments include Arduino [6], in which the programmer must connect wires to pins to join hardware modules. Programming in Arduino is done in a C-like language which provides direct access to the pins. Booth et al.’s study of end-user developers in Arduino suggests that most critical errors involve “incorrect circuit construction” [8]. The approach taken in Ansible and similar programming environments hides circuit construction and may prevent these errors.
2.2 Widgets for physical computing systems

Many widget collections have been created and described for physical computing. In most cases, the widgets are simple input devices, such as buttons, sliders, or knobs. More advanced widgets might include accelerometers, screens, and other components. The differentiating factor in widget sets tends to be the mechanism for creating interaction; examples include pneumatics [27], light [22], sound [23], electronic components [14], and large mechanisms [4]. Widgets in Ansible are implemented using off-the-shelf electronic components, but the selection process in Ansible could be added to widgets with other mechanisms. While this is not a contribution of the Ansible system, a note of potential interest is that we have added a single-board computer to the widget. This computer supports the selection process as well as other processes on the widget itself.

There are other potential selection modalities that may provide paths for future work. RapID [24] creates interactive physical systems by embedding RFID tags into otherwise passive objects, such as pieces of wood. The embedded tags can be used to track both the position and—more importantly—the identity of objects in a room or laboratory. Phybots [15] prototypes locomotive robotic widgets using visual markers captured by a topdown camera. With systems such as these, a new select-to-edit modality could be implemented where shaking or tapping a physical widget or object is used to select it.
Chapter 3

Implementation

3.1 Programming widgets in Ansible

The main contribution of our work is an intentionally simple process for selecting physical widgets during programming.

To select a widget, the programmer performs a physical action. The three actions or widget selection modalities that Ansible supports are: pressing a button on the widget, shining a light on the widget, and waving a magnet over the widget. Some strengths and weaknesses of each selection method are highlighted in Table 3.1.

Selecting a widget brings up its code in a simple IDE on the programmer’s laptop or workstation. When the programmer is done editing the code, they click a save button and the widget is updated with the new functionality.

While editing code, the programmer can also select another widget (using the same physical action as before) to insert its name into the code at the current cursor location. This is helpful when referring to other widgets in code. As an example, consider a programmer who wants a button press to toggle an LED across the room (see Figure 3.1). While programming the button widget’s click handler, they reach the part in the code where they need to toggle

Table 3.1: Strengths and weaknesses of three widget selection methods implemented in Ansible.

<table>
<thead>
<tr>
<th>Method</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>no tools required</td>
<td>need physical access</td>
</tr>
<tr>
<td>Light</td>
<td>works at a distance</td>
<td>widget must be visible</td>
</tr>
<tr>
<td>Magnet</td>
<td>no visibility needed</td>
<td>need close access</td>
</tr>
</tbody>
</table>
the LED. Rather than having to remember or look up the LED widget’s name, they shine a laser pointer at it, and the LED’s name automatically appears in their code.

### 3.1.1 Widget behaviors

In Ansible, programmers have four ways to create widget behaviors: widget event handlers, an initialization function, a widget loop function, and shared global state. Event handlers include behavior such as `on_press()` for buttons. The initialization and loop functions mimic the same functions on the Arduino platform [6], in that initialization is called once (when the widget is turned on or when code is saved) and the loop function runs continuously. Shared global state can be accessed and modified by every widget in the system, and widgets are able to set an event handler to be notified when the shared global state changes. The programmer can also define local fields and methods on the widgets.

For clarity, we give two examples to illustrate programming in Ansible. The following code implements a system in which pressing and holding `button63` (for example) turns on LED `led14` until `button63` is released, by using widget event handlers. The code lives on `button63`.

```python
def on_press(self, global_state, widgets):
    widgets.led14.on()
```

Figure 3.1: A programmer uses the laser pointer to select an LED widget and thus insert its name in the button widget’s code.
def on_release(self, global_state, widgets):
    widgets.led14.off()

    Ansible also supports setting handlers in the reverse direction. In the following example, the code resides on the LED widget rather than on the button widget. On setup, the LED registers itself for the button’s on_press and on_release events:

def setup(self, global_state, widgets):
    widgets.button63.on_press = self.on
    widgets.button63.on_release = self.off

    We note that the novelty of Ansible is not this event-driven approach to programming, but that the programmer can physically select a widget either to edit its code or to insert its name in the code.

3.2 Ansible architecture and protocols

The following section is ancillary to the main contribution—namely, physical selection—and is not in itself a contribution, but we do want to provide sufficient detail on how these widgets work.

3.2.1 Hardware

Ansible uses a server/client architecture in a star topology with many widgets connected to a single server, as shown in Figure 3.2. The server runs on a laptop or desktop workstation with a monitor, keyboard, and mouse for use during programming. Each widget is a client, and all client-to-client communications go through the server. This simplifies client code, and in practice we found that the bottlenecking was negligible and did not cause any problems. Other topologies—such as peer-to-peer—would also have worked, with a different set of advantages and disadvantages.
A widget consists of three parts as is also shown in Figure 3.2: a component, a selection sensor, and a single-board Raspberry Pi Zero W computer running Linux. The component is the interactive part of the widget, such as a button, screen, or motion sensor. If the component is an input device, it sends events to the widget computer; if it is an output device, the widget computer sends output to it. Electronics implementation details concerning component-computer communication are hidden from the programmer. The selection sensor detects when the programmer has selected the widget and sends that selection data to the widget’s computer (see the next section for more detail on the communication protocols). The widget computer forwards the selection data to the server. It also runs the programmer’s code for the widget and manages all communication with the server, the component, and the selection sensor.

While the following is also not a contribution, we note that because the widget contains a single-board computer running Linux, the widget can handle a great deal of processing locally. This minimizes unnecessary server load and network traffic, which could be significant—for example, audio/video processing on the widget itself would save bandwidth on data that wouldn’t need to be sent to the server. The computer also enables Ansible...
widgets to do more work than earlier physical widgets such as Phidgets [10, 19] or .NET Gadgeteer devices [28].

The Ansible server keeps track of each client widget by a unique identifier, which is paired with the widget’s name. When a widget is selected, its code is edited on the laptop or workstation running the server; when editing is complete, the server pushes the new code back to the widget. The server also tracks which widgets have registered for events on other widgets.

Widgets are assigned default names when prepared for use in Ansible, but they can be selected for programming without the programmer having to know the names, which can be convenient.

3.2.2 Protocols

The communication of inter-widget behavior—executing methods and reading properties on remote widgets—is as follows (see Figure 3.3). We will not cover details regarding timeouts, error recovery, exceptions, or other communication problems, since those are ancillary and are not particularly interesting.

A widget contains a stub used to access remote widgets. This process is shown at the top of Figure 3.3. When a programmer writing code for widget A accesses a field x on widget B using an expression like \texttt{widgets.B.x}—where \texttt{widgets} indicates that B is remote from A—the stub is invoked instead. It sends a message to the Ansible server requesting access to field x on B. If the server is not aware of B, A is notified; if it is, the server sends B a request to access field x. When the server receives the response from B, it forwards the response to the original widget A.

Setting a field is similar. The programmer writing code on A sets field y on remote widget B using \texttt{widgets.B.y = r} (where \texttt{r} indicates the right-hand value). The stub sends a similar message to the Ansible server, this time including \texttt{r}. The server relays the request to
Figure 3.3: Inter-widget behavior flow. Only the server maintains a list of connected widgets.

To call a method on a remote widget from widget A, the programmer uses an expression like `widgets.B.m(p)`, where `m` indicates the method and `p` indicates the parameters to the method call. The stub sends a message to the Ansible server requesting execution of `m` on B, with `p` as parameters. The server relays that message to B, which then executes `m` if it exists—passing in `p`—and returns the response. The server then relays the response, even if it is a null value, to the original widget A.

To make widget A listen for an event `e` on a remote widget B, the programmer registers a handler on A by using an expression like `widgets.B.on_e = fn`, where `fn` indicates a local callable. The stub registers A’s handler for event B.on.e with the server. When `e` happens on B, B notifies the server. The server notifies all the registered widgets of the event, and each of those widgets (including A) then executes its local handler for event `e`. This is shown at the bottom of Figure 3.3.
One consequence of Ansible’s choice of star topology is that only the server needs to maintain a list of all the widgets in the system. Peripheral widgets forward all remote widget requests through the server, and the server must determine whether or not the requested remote widget exists. This means that Ansible widgets do not in themselves know which other widgets are connected to the system. Even if the Ansible IDE implemented widget name auto-completion, which it does not, the widgets would not need to maintain a list of connected widgets because Ansible code is always edited on the server.

3.3 Ansible implementation

In order to solve the selection problem, we needed infrastructure that would allow us to implement various selection modalities in a real, working system. To that end, we will describe in more detail how we built Ansible.

Ansible can be seen as an updated implementation of Greenberg and Fitchett’s Phidgets [10]. Rather than using COM objects, USB cables, and C++, Ansible uses wireless networks—allowing more flexible widget placement—and Python. In the following sections, we describe our updated implementation in detail. The software implementation leverages unique features of Python as an interpreted language that can dynamically reload modules at runtime. We also overload method and field access operators in Python. The hardware implementation uses off-the-shelf components for ease of construction.

3.3.1 Software

Ansible is written in Python and consists of a server program that runs on the programmer’s laptop or workstation and a client program that runs on each widget’s computer. Widgets communicate wirelessly with the server.

The widgets and server must know their respective IP addresses to be able to communicate. We use a UDP broadcast on a known port to establish a connection and then use TCP to exchange IP addresses and all future messages. Message data is sent using JSON
payloads, consisting of a command ("select widget," etc.) and a body (containing parameters and other data). The server keeps a list of connected widgets in which it tracks each widget’s name, GUID, and IP address.

When a widget is selected, it sends a "select widget" message to the server with the widget’s code in the body. If no other code is being edited on the server, the server displays the widget’s code for editing; if the server is already in code-editing mode, it inserts the name of the widget at the current location of the text cursor.

After the programmer finishes editing the widget’s code, the server sends a copy of the new code back to the widget, which saves the code and reloads it dynamically using `importlib.reload()` from Python’s standard library (see [3] for additional details).

**Referencing remote widgets**

To support referencing remote widgets in code, we overload the built-in Python getter/setter methods `__getattr__()` and `__setattr__()` to create a stub which allows the programmer to use standard, well-understood dot notation such as `widgets.button6` while avoiding less ergonomic syntax such as `widgets[‘button6’]` and `widgets[‘button6’][‘func’]()`. Also of note: the server, not the widget, resolves remote widget field and method references.

For setting local handlers to watch events on remote widgets (e.g., something along the lines of `widgets.button6.on_press = mylocalfunc`), we store a map pairing the local widget’s name with the callable itself and then generate a key. We register that key with the server so the server knows which widgets are interested in the event, and so the local widget can use the key to access the intended callable when the remote event is triggered.

When an input event—such as a button press—happens on a widget, that widget notifies the server. This notification is sent regardless of what the programmer does on that widget for that event.
Figure 3.4: Three of the six implemented Ansible widget types. From left: a LED widget, a button widget, and a proximity sensor widget. Each widget contains an interactive component, a selection sensor, and a Raspberry Pi Zero W computer.

**Physical widget selection**

This is the core of our contribution, namely enabling select-to-edit functionality using physical, demonstrative modalities. The selection modalities supported include: a button, a light, and a magnet.

Selection using a button simply involves waiting for a change in the state of the GPIO pin to which the button is connected on the Raspberry Pi, using the `gpiozero` library.

Selection using a light is done using a thresholding algorithm on a photometer. We sample the sensor at 100 ms intervals and keep a baseline moving average over the last 40 samples. If the new sensor reading is more than four times the baseline average for the light, and if it has been more than 2.5 seconds since the last selection, we trigger a selection event. The photometer code was implemented using the Adafruit TSL2561 Python module.

Selection using a magnet involves the same process as selection using a light, but with a magnetometer instead of a photometer. The magnetometer code was implemented using the Adafruit LSM303 Python module.

**3.3.2 Widget hardware**

Three Ansible widgets are shown in Figure 3.4. Each widget consists of a Raspberry Pi Zero W single-board computer, a selection sensor (a button, a photometer, or a magnetometer), and a component (a button, LED, accelerometer, proximity sensor, screen, or speaker). The
Raspberry Pi is located beneath the top plate of the widget housing and is largely obscured by the housing. The wires extending behind each widget are USB power cables; we also implemented widgets using built-in batteries.

Beyond the pictured LED, button, and proximity sensor widgets, we also built accelerometer, speaker, and screen widgets, which are not shown here but can be seen in the example applications section. In total, we built widgets of all six types using a photometer for the selection sensor, along with a smaller number of widgets using either a magnetometer or a button for the selection sensor.

The widgets are housed in a custom casing made of laser-cut black plexiglass assembled with nylon standoffs and nuts and bolts. In Figure 3.4, the exposed circuit boards on the LED and button widgets are the selection sensors; one of the exposed circuit boards on the proximity sensor widget is the selection sensor, and the other is the proximity sensor component.

The buttons are 12mm waterproof momentary push button switches. The light sensors are Adafruit TSL2561 boards. The magnet sensors are Adafruit LSM303 boards. The LEDs are 3mm clear LEDs, mounted with steel LED holders. The accelerometers are Adafruit LIS3DH boards. The proximity sensors are VL6180X boards. The screens are small OLED displays using I2C. The speaker is a Pimoroni Speaker pHAT.
Chapter 4

User Study

We conducted a qualitative, inductive user study intended to explore how using a laser pointer to select widgets changed the experience for programmers compared to using a more traditional menu system. We had participants perform a fixed task for a fixed period of time and then asked them open-ended questions to elicit their experience with the task.

4.1 Methods

We conducted the study with twelve participants to assess what the experience of using Ansible was like and to see what qualitative differences there were between selecting widgets with the traditional system and selecting them with a laser pointer, using Ansible’s light sensor modality. We recruited participants directly and via announcements in classes.

The twelve participants—six in the control group and six in the experimental group—were college students with Python experience. Sessions lasted 45 minutes or less and participants were compensated $15 for their time. Each participant gave informed consent before participating in the study. All recruitment scripts, consent forms, and study plans were approved in advance by our Institutional Review Board (IRB). No adverse conditions were reported.

In order to motivate participants to perform at their best, they were told at the beginning of the session that the top performing participant would receive an additional $40 in compensation for their excellent work. After all participants completed the study, we awarded one participant the additional $40 for their thorough and efficient work.
We explained to each participant what Ansible was and demonstrated how to use the system—specifically, how to select widgets and how the API worked. The control group used a modified version of Ansible with a GUI where the participant was able to click on buttons in a sidebar menu to select widgets. The experimental group used Ansible’s light selection modality with a laser pointer to select widgets.

We gave each participant a short familiarization task where they were instructed to program a button so that when it was pressed, a nearby LED would toggle on and off. We also answered questions during this task.

After the familiarization task, we gave the participant instructions to build a small game using seven widgets—buttons, LEDs, and a speaker—that were spread throughout the room. In the game, the player presses a start button, which initiates a timer and turns two LEDs on. The speaker indicates the timer status (a beep each second, for example). When the player presses one of the two other buttons, the paired LED turns off. The player’s goal is to use the buttons to turn off both LEDs before the timer runs out. During this task, one member of the research team sat in the room and took notes on how the participant used Ansible.

At the end of the study, we had the participant fill out a short questionnaire which asked them to rate the difficulty of widget selection and to answer a small number of open-ended questions:

1. On a scale of 1 to 10 (with 1 being easy and 10 being difficult), how hard was it to use the light [or the buttons, for the control group] to select widgets? Why?

2. Tell us a little about your experience maintaining the mapping between your code and the physical widgets.

3. If we arbitrarily chose one of these widgets and asked you what its name was, how confident would you be in your answer?

4. Anything else you’d like to say about your experience here today?
We note that the numeric scale in question 1 was intended to help the participants collect their thoughts and was not intended for quantitative analysis.

To analyze the data, four members of the research team read through participant responses and reviewed notes taken during the programming sessions. They then met to identify themes in the data and to select representative quotes.

4.2 Results

In exploring how using a laser pointer changes the experience of programming widgets, we identified several recurrent themes in participant behavior and responses. Participants E1–E6 were in the experimental group and used the laser pointer; participants C1–C6 were in the control group and used the buttons in the sidebar menu.

4.2.1 Mapping widgets to code

The predominant theme that emerged was that using the laser pointer eliminated the need to maintain a mapping—physical or mental—between physical widgets and their names in code.

The control group participants repeatedly talked about those mappings. They commented on the initial need to map widgets to code via trial and error. Some participants developed hand-drawn maps, as C3 mentions: “I used pencil and paper and initial experiments to map the buttons.” Similarly, C4 said: “I drew a diagram with how the widgets were layed [sic] out on the table & wrote the numbers next to them. To find out the mapping I first saw which widgets had button press methods & used those to test the non-button widgets.”

Other participants, like C6, maintained their mapping digitally: “I had to use comments in the code to keep track of what was what, since there were no on screen reminders. Once I figured out what mapped to what the code was easy to write.”

And others, like C1, relied on a mental mapping: “It was difficult keeping a mental model of what mapped to what especially since you had to go through a two step process to look at the other widgets.”
In the experimental group, in contrast, five of the six participants said the system was easy to use or intuitive, and none of the six talked about mappings at all. Participant E5 said: “I feel like it was generally really intuitive (especially with not having to remember the names of the different widgets in order to access their functions).” In a similar vein, E2 said: “using the light to select the widgets was very easy and very nice.” Continuing along the same lines, E6 said: “the light worked really well to select the widget I wanted.”

A note of interest was that one participant (E3) mentioned that while the light was more helpful at the beginning, they then began to build a mental mapping that ended up being faster and easier for them: “The light helped more at the beginning because I didn’t need to remember the name of the widget, but as I worked more and more I memorized the widget names anyways and was familiar enough to the point that I didn’t need to see the widget physically to work with it.” For some programmers, a mental mapping may be more effective, at least at small, manageable scales.

4.2.2 Confidence in identification

In the questionnaire we asked: “If we arbitrarily chose one of these widgets and asked you what its name was, how confident would you be in your answer?.”

The control group generally felt they would be highly confident, as long as they had their external mapping list. Participant C2 said: “After going through this, if I can look at the screen where I named them, I am 100% certain”. C3 said: “Very confident, provided I had my list.” C4 said: “Maybe a 3 out of 10 without looking at my map. If I had my map 10 out of 10.”

Overall, the experimental group’s confidence levels were lower than the control group’s. Participants who had manually renamed widgets, however, noted that they were confident—but only for the specific widgets they had renamed. Participant E2, for example, said: “Not very confident, I would know the ones I renamed by placement, but that’s it.” E3 said: “I
Table 4.1: Participants’ use of the light selection modality.

<table>
<thead>
<tr>
<th>Participant ID</th>
<th>Number of selections with laser pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>30</td>
</tr>
<tr>
<td>E2</td>
<td>20</td>
</tr>
<tr>
<td>E3</td>
<td>30</td>
</tr>
<tr>
<td>E4</td>
<td>20</td>
</tr>
<tr>
<td>E5</td>
<td>21</td>
</tr>
<tr>
<td>E6</td>
<td>32</td>
</tr>
</tbody>
</table>

renamed some of the widgets, and those I would be confident in naming. The widgets that I
didn’t use, however, I do not know the names for.”

These results suggest that pointing to select does meet Ansible’s goal of freeing the
programmer from having to maintain a mapping between widgets and code, allowing them
to focus more on the problem at hand.

4.2.3 Frequency of selection

We reviewed the video of the user study and noted the number of times each of the experimental
participants used the laser pointer to select a widget. The results (see Table 4.1), show that
participants selected widgets using the light selection modality a mean of 25.5 times (n = 6,
standard deviation = 5.22). Given that these were first-time users, these numbers show that
the system works and implies that it is easy to use.
Chapter 5

Example Systems

We built three example systems to demonstrate the range of applications Ansible makes it easier to create, the variety of widget types Ansible can support, and the different physical scales of systems that can be created. All three are shown in the accompanying video.

5.1 Pill bottle monitor

The pill bottle monitor for the elderly tracks when the medicine cabinet door (see Figure 5.1a) is opened or closed (which sends an email to the caregiver) as well as when the pill bottle (see Figure 5.1b) is taken from the shelf or put back (which also sends an email to the caregiver). If the pill bottle isn’t moved by a certain deadline, the system texts the caregiver. We used a proximity sensor widget and an accelerometer widget for this small-scale (cabinet-size) system. This system took approximately ten minutes to program.

(a) The accelerometer sensor widget is attached to a cabinet door.

(b) The pill bottle is on top of the proximity sensor widget inside the cabinet.

Figure 5.1: Pill bottle monitor.
The accelerometer widget’s `loop()` handler was programmed to calculate the magnitude of the sensor’s motion every 100ms. If the magnitude was greater than or equal to 9.60 (the baseline the accelerometer sensor reported at rest) and the widget hadn’t sent an email in at least 1.5 seconds (tracked via a timer variable), it sent an email to the caregiver and set the timer variable, which it incremented on each pass through the loop. As a sample, we present the code for the `setup()` and `loop()` handlers:

```python
def setup(self, global_state, widgets):
    self.timer = 15

def loop(self, global_state, widgets):
    if self.magnitude() >= 9.60 and self.timer > 15:
        self.send_email('myemail@gmail.com', 'Cabinet door moved', '')
        self.timer = 1

    self.timer += 1
    time.sleep(0.1)
```

The proximity sensor’s `loop()` handler was programmed to poll the sensor’s distance value (5–255mm) each 100ms. If the distance value was above the threshold of 60mm and the bottle_on flag was set to true, the widget emailed the caregiver and set bottle_on to false. If the distance value was below or equal to 60mm and bottle_on was false, the widget emailed the caregiver, set bottle_on to true, and saved the current time. The `loop()` handler also checked to see the elapsed time (comparing the current time against the saved time), and if that elapsed time was large enough and bottle_on was true, the widget sent a text (via email) to the caregiver.
5.2 Room-size adventure game

The room-size adventure game is an aural piece of interactive fiction. Button widgets are placed on each of the north, east, south, and west walls of a room. A speaker widget (see Figure 5.2a) reads directions and instructions out loud. For example, “You are standing in a long, ancient hallway made of red stone.” To move around in the game, the player walks around the real room and presses the cardinal direction buttons on the walls (see Figures 5.2b and 5.2c). A screen widget shows the player the name of the room they are in. This game demonstrates a room-scale system and took approximately twenty minutes to program.

The start button widget was programmed to set the player’s location in the global state. The other four button widgets were programmed to check the current location and then update the global state with the player’s new location (if the player was able to move in that direction) or with a custom message (“you can’t go that way,” for example).
The speaker widget’s `on_global_state_change()` handler was programmed to check for location changes or messages and to read the new room description or message, respectively. We originally used the `espeak` command to generate the audio, but voice quality made it difficult to understand, so we generated canned MP3s using `say` and played those instead. We present a sample from the speaker code:

```python
def on_global_state_change(self, global_state, widgets):
    if global_state.get('moved'):
        room = global_state.get('room')

    if room == 'hall':
        widget.play_mp3('hallway.mp3')
    elif room == 'archive':
        widget.play_mp3('archive.mp3')

    global_state.update({
        'moved': False,
    })
```

The screen widget’s `on_global_state_change()` handler was programmed to check for location changes and update the screen with the new room name if the location changed.

### 5.3 Home door lock checker

The home door lock checker is a reminder system to help the user remember to check that their home’s doors are locked each night. In the master bedroom are a set of LED widgets, each of which corresponds to one of the doors. At a specified time each night (see Figure 5.3a), those LEDs turn on automatically, reminding the user. Next to each door is a button widget (see Figure 5.3b); when the user has checked that door, they press the button, which
(a) The lights turn on at the specified time.

(b) The user checks the lock and then presses the button on the widget attached to the wall.

(c) When the user presses a button widget, the associated light widget turns off its light.

Figure 5.3: Door lock checker.
then turns off the corresponding LED in the master bedroom (see Figure 5.3c). This checker demonstrates a house-scale system with many widgets and took approximately two minutes to program.

Each LED widget was programmed to check the time every second and, if it was the appropriate time, turn the LED on. It also set a flag at that point so that it wouldn’t keep trying to turn the LED on. We present sample code for one of the LED widgets:

```python
def setup(self, global_state, widgets):
    widget.activated = False

def loop(self, global_state, widgets):
    now = datetime.datetime.now()
    # Check to see if it’s 7:00, and if so, turn on
    if now.hour == 19 and now.minute == 0 and not widget.activated:
        widget.on()
        widget.activated = True
    time.sleep(1)

    # Each button widget was programmed to turn off the paired LED when the button was pressed.
```
Chapter 6

Conclusion

Ansible demonstrates an intuitive select-to-edit process for physical widgets (where “selecting” can involve shining a light on a widget, waving a magnet at it, or pressing a button on it) that relieves the programmer from having to maintain a mental mapping between widgets and their names in code. The example systems demonstrate some of the variety of functions, spatial scales, and widget types supported by Ansible.

These results are significant because they may simplify interactive system implementation distributed over collections of physical widgets, perhaps especially large collections. These kinds of distributed physical systems may become more common as we move from the post-desktop era into a post-smartphone era.

One explanation for why Ansible reduces mental load for programmers is that demonstrative gestures—like pointing—are a core part of how people communicate. Adding a laser pointer or a magnet to this process seems to build on these gestures. One of the themes found in the user study was the ease of use and intuitive nature of using the laser pointer. In our experience building example systems, the names ended up mattering so little that we stopped renaming the widgets and just used their generic names assigned on initialization (such as \texttt{widget1}, \texttt{widget2}, etc.). For longer-term maintenance, however—reading the code weeks or months later—or when more than one person works on a system, widget names do matter.

The Ansible system as implemented has several limitations, two of which we mention here. First, the star network topology has a single point of failure. If the server fails or
becomes congested, the entire system will go down. This could be problematic for systems with hundreds of widgets. Second, the on-board computers require more power, which constrains widget placement if widgets are wired or duration if widgets run on batteries.

Three weaknesses of our evaluation of Ansible are that we did not evaluate long-term use, we did not use it with tens or hundreds of widgets, and we did not conduct studies at scales larger than a room. It simply was not feasible to conduct the first two kinds of studies. Studies over a period of months or years using tens or hundreds of widgets may however lead to important new insights about assisting programmers in maintaining the mapping between physical widgets and the widgets’ names in code. We also note that our study did not test participants’ ability to recall names, because our work on Ansible was not focused on that ability, but rather on the point-to-select idea.

Future work might consider location-based selection over a wide area by placing markers at widget locations on a map on a screen. The sensor maps in [19] might guide this approach. GPS units added to Ansible widgets would allow them to report their location, which could then be used to populate the map. This selection process could be helpful when widgets are spread over several kilometers, especially if they are moving. Research into other selection methods with different strengths may prove fruitful as well.
References


