10 Building Adaptive Touch Interfaces—Case Study 6

Daniel Buschek and Florian Alt

10.1 Motivation for Adaptive, Probabilistic Touch Interfaces

Generally speaking, an adaptive user interface (UI) is one that does not look and work in the same way for everyone or at all times. In contrast, static UIs stay fixed across users and contexts. The idea of building adaptive UIs promises many benefits for users by addressing specific characteristics of interaction behavior which arise due to 1) the individual user and 2) the current context of use. For example, adaptations often aim to improve efficiency and effectiveness of interaction under *varying* conditions, such as supporting accurate smartphone touch input with different hand postures (e.g., thumb vs. index finger, left vs. right hand [Buschek and Alt 2017]) or in different situations (e.g., sitting vs. walking [Goel et al. 2012, Musić and Murray-Smith 2016]).

Adaptation seems particularly intriguing for mobile devices since these are used in a large variety of everyday situations (e.g., at home, at work, on the go, on public transport; also see Sarsenbayeva et al. [2017]). Moreover, smartphones are often seen as highly personal devices, linked to one specific user. Thus, it seems relevant and useful to investigate how such a device might adapt to an individual user and his or her specific capabilities, habits, preferences, and so on.

These ideas have caught increasing interest by many researchers over many years (e.g., see Browne et al. [1990], Wahlster and Maybury [1998], Calvary et al. [2003]). In particular, they have been picked up by people working at the intersection of human–computer interaction (HCI) and machine learning (ML). Adapting UIs presents a prime example for opportunities arising from the combination of expertise in both these fields: ML is required, for instance, to model user behavior, to detect changes in contexts and behavior, also using sensor data, and to optimize the UI based on derived information. On the other hand, HCI expertise is important to smoothly and usefully integrate adaptations into user interactions and actual practices of use, and to inform many surrounding questions, for example, about interface consistency, user control, and (mixed) initiative [Horvitz 1999].

It is at this intersection that *probabilistic* approaches become highly relevant and useful, since reasoning based on human input as well as incomplete and noisy sensor data has to deal with uncertain information (cf. Williamson [2006]). Several research projects thus explicitly address handling uncertainty in (adaptive) touch interfaces (e.g., see Schwarz et al. [2010, 2011], Weir et al. [2012, 2014], Bi and Zhai [2013]).

To gain a more intuitive understanding from a practical developer perspective of why a probabilistic approach is practically useful, consider the task of implementing from scratch a mobile touch graphical user interface (GUI) with two sliders. How should we decide, for a given sliding trajectory of the finger, which slider the user wanted to activate? Easy—the closest one! But what if the finger trajectory is "wiggly" and touches both of them (e.g., due to mobile use on a shaky bus ride)? And how do we measure "closest" exactly? We could come up with some custom "scoring" measure but that might be inconsistent with the implementation that our colleagues chose for other parts of the software (e.g., other GUI widgets). In summary, every time we have to think about manually "scoring" user input with regard to some decision-making, we are potentially struggling with a fundamentally deterministic setup for input that we would actually like to treat probabilistically—and hence we could benefit from a principled probabilistic approach.

Examples of concrete user- and context-specific adaptations in (mobile) touch interfaces are abundant in recent HCI work: They include, for instance, adapting keyboards to walking [Goel et al. 2012] and individual users [Findlater and Wobbrock 2012], as well as ways of holding the device [Goel et al. 2013, Buschek et al. 2014], or multiple such factors at once [Yin et al. 2013]. Other work corrected touch points based on user-specific targeting behavior [Weir et al. 2012, Buschek and Alt 2015], or supported touch input for users with specific motor impairments [Mott et al. 2016].

Many such projects build research prototypes for evaluation in a user study, but not for further use or even a "productive deployment." Thus, both conceptual and practical aspects of development and engineering remain an underexplored part of realizing the vision of user- and context-adaptive touch interfaces. Yet these aspects are of high practical importance for interactive system developers who want to benefit from digital signal processing and ML. In the spirit of this book, this case study (CS) therefore discusses the authors' probabilistic user interface

(*ProbUI*) framework [Buschek and Alt 2017] as a concrete example for supporting developers in building adaptive mobile touch interfaces.

10.2 Three Key Challenges for Developing Adaptive Touch Interfaces

Overall, *ProbUI* is motivated by three key challenges for developers: 1) Specifying complex input behaviors (here: touch gestures), 2) recognizing and distinguishing said behaviors, and 3) handling and reacting to such input under uncertainty. Figure 10.1 presents an overview. Before we take a closer look at *ProbUI*, let us first examine these challenges in more detail.

10.2.1 How to Describe Complex Touch Behaviors and Integrate Them into UIs?

Many adaptive and/or probabilistic touch UIs address not only simple taps, but also more complex touch behaviors, such as gestures (e.g., swiping, scrolling). For the developer, this raises the question of how to describe and integrate such gestures into a UI, in particular when simultaneously considering probabilistic reasoning and adaptation.

Existing Options for Developers In general, there are several main approaches to integrating gestures (i.e., recognizers) into touch UIs: First, developers could rely on *pre-defined gestures* from an application programming interface (API) or library. In this case, implementation and integration might be trivial, yet it is not clear how well such a library would also support probabilistic reasoning and UI adaptation.

Second, developers could *implement their own system* from scratch. While this gives them full control, and might be preferred by developers who are themselves confident and experienced with probabilistic reasoning, it might often result in significant increases in time and effort spent on the project. Some support could be provided by making use of general probabilistic inference frameworks, such as *Infer.NET* [Minka et al. 2014].

Third, *programming-by-demonstration* presents an alternative for setting up touch gesture models (e.g., Lü and Li [2012, 2013], Lü et al. [2014], Li et al. [2014]). In this case, developers record gesture examples with a dedicated toolkit, and possibly refine them, ideally to then generate compatible gesture recognition code. A downside of this approach is that developers have to actively record data with external tools, the results of which then need to be integrated into the application/UI code.

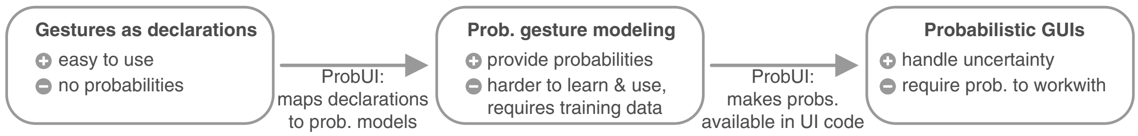

Figure 10.1 Overview of the three key areas and related strengths ("+") and challenges ("−"). *ProbUI* for the first time links these three areas in a single pipeline in one framework, thus supporting developers in 1) specifying input behavior, 2) recognizing said behavior under uncertainty, and 3) reacting to it, for example, with GUI adaptations, again considering uncertainty.

Finally, some researchers have suggested *declaration* as an easy-to-use approach to gesture specification. Here, developers use a simple language to write down the desired touch behavior/gestures. For example, one might simply say "swipe right" to denote said behavior. Actually proposed declarative languages are not that verbatim but rather stay on a level of abstraction that renders them similar to regular expressions. An example is the *Proton* language for multitouch gestures [Kin et al. 2012a, 2012b]. Earlier related work includes *GDL* [Khandkar and Maurer 2010] and *Midas* [Scholliers et al. 2011], which also offered a rule-based reasoning system. *ProbUI* follows this declarative approach.

Declaration—Strengths and Challenges The main strength of the declarative approach is its ease of use for the developer. In particular, declaration offers concise and readable specifications, often directly embedded into the code (e.g., as a string parameter), without the need to switch from the development integrated development environment (IDE) to external tools. However, the original declaration approach does not lend itself well to handling variations in user behavior (e.g., different finger trajectories for the same swipe command), as well as probabilistic reasoning and handling uncertainty in general (cf. *Proton's* discussions [Kin et al. 2012a, 2012b]).

As we will see in Section 10.3, *ProbUI* uses declarations and rules for their ease of use for the developers, yet treats them as input to an algorithm that automatically derives probabilistic models to address these challenges.

10.2.2 How to Recognize Touch Behaviors Probabilistically?

The second key challenge is to recognize the defined behavior during use, in particular in a *probabilistic* and adaptive UI concept. Here, developers face the question of how to derive probabilities for users' touch input. This question is obviously linked to the first one; again, there are several options.

Existing Options for Developers As a first option, developers could simply cut the probabilistic treatment and recognize gestures with one of the easy-to-integrate declarative approaches from the previous section. However, they then miss out on handling uncertain user input, continuous feedback under uncertainty, and related mechanisms of UI adaptation (also see e.g., Williamson [2006], Schwarz et al. [2015], Buschek and Alt [2017]).

Second, developers could *build their own gesture models* from scratch, for example, using a ML framework. This possibly incurs costly data recording if no fitting dataset already exists, plus time spent on development, debugging, and evaluation.

Third, developers could use some of the above mentioned *programming-by-demonstration* tools which generate probabilistic gesture recognition models (e.g., see Lü and Li [2012, 2013], Li et al. [2014]). Again, the downsides here involve the need for external tools and data recording.

Probabilistic Reasoning—Strengths and Challenges The strengths and challenges of probabilistic reasoning for gesture recognition are inverse to the ones discussed for declaration: Probabilistic approaches are generally more difficult to setup and integrate into UIs, but they offer attractive benefits when for handling variations in users' input behavior, uncertain and noisy sensor data under varying contexts, unclear user intentions, and so on.

As we will see in Section 10.3, *ProbUI* takes the developer's declarations (plus UI specifications) to automatically derive a simple yet consistent probabilistic model. This novel pipeline merges the benefits of both approaches, namely ease of use for setting up gestures with declarations, and the power of probabilistic input interpretation during interaction.

10.2.3 How to Handle and React to Uncertain User Behavior Information?

Finally, assuming probabilistic input, how can UIs make use of it for the benefit of the user? This is the third key question that developers of probabilistic and adaptive (touch) UIs have to respond to.

Existing Options for Developers In a very simple approach, probabilistic input events could be *"thresholded" to treat them deterministically*, thus ignoring and losing the uncertainty information. Again, one could also implement a *custom mapping* of probabilities (e.g., from a gesture recognizer) to some UI variables (e.g., transparency of a button linked to gesture shortcut probability). Another use of such probabilities are *custom rules*, such as if-else statements that reach a decision based on probabilities. The larger underlying question here often relates to how exactly to treat these numbers. Earlier research projects informally outlined, for example, selection rules (e.g., see selection of sliders in Schwarz et al. [2010] or probabilistic representation of scrolling in Schwarz et al. [2015]).

Despite such case-to-case implementations in some examples, the related work provides many ideas, concepts, and overarching *frameworks for consistently handling probabilities in UIs* (e.g., see Mankoff et al. [2000a, 2000b], Schwarz et al. [2010, 2011, 2015]). A key concept is a so-called "mediator," that is, a software component that takes in all probabilistic information, requests from UI elements, and other data, to reach a global decision (e.g., which button to activate) and to do bookkeeping work (e.g., cancelling intermediate visual feedforward/back once the decision has been made).

Probabilistic GUI Frameworks—Strengths and Challenges Probabilistic GUI frameworks' strengths lie in their support for dealing with uncertainty in user input behavior and context information. This renders them highly relevant for adaptive UIs. However, the challenge of most such existing frameworks is that they require other software components to provide them with these probabilities in the first place. In other words, these frameworks do not derive probabilities themselves. Thus, developers have to manually hook them up to, for example, the probabilistic gesture recognizers mentioned before. This generates manual development effort.

ProbUI improves on this by establishing a pipeline from 1) gesture declaration over 2) gesture recognition to 3) interpretation in one framework: When declaring gestures and rules, developers can already implement callbacks for handling resulting events. In addition, developers can easily and directly access any probabilistic information at any point in the application/UI code (e.g., when implementing a UI widget class). Following related work, a mediator object takes care of the appropriate global decision-making processes at runtime during interaction.

10.3 The *ProbUI* Framework

Several aspects of the *ProbUI* framework have been motivated already when describing the key challenges and related options for developers in the preceding section. Building on this background, this section now introduces *ProbUI* in more detail.

ProbUI addresses the described set of challenges with a conceptual framework that merges benefits from three areas of related work by offering a novel development pipeline. This conceptual framework is also implement as an Android library for practical use (see Buschek and Alt [2017]).

10.3.1 Overview

In brief, this pipeline first allows developers to specify touch behavior per UI element with a declarative language. For example, a developer might assign a tap and both left and right swipes to a multi-functional button. Next, *ProbUI* takes the developer's specifications to automatically derive a basic probabilistic model (simple hidden Markov models or "HMMs" [Rabiner 1989, Barber 2012]). In a way, this probabilistic GUI representation brings some of the ideas of touch modeling in adaptive keyboards (e.g. see Chapter 7 CS 7.3.2) to GUIs in general, beyond keyboards. Finally, during use, *ProbUI* continuously infers the user's intended behavior and target. It does so in a probabilistic manner. As a result, developers can access and utilize probabilities about behavior (*What is the user doing?* E.g., swiping) and targets (*Which UI elements is the user using?* E.g., button X; or *How likely is it that the user really wanted to trigger this button?*). This information is useful, for example, to implement live feedback and adaptations (e.g., reacting to left- vs. right-handed use, see examples in Section 10.4.4). Figure 10.2 visualizes a development example using the framework.

Next, we explain the core components of the framework in more detail, before discussing concrete development examples step-by-step in Section 10.4.

10.3.2 *ProbUI*'s Modeling Language (PML)

ProbUI introduces a simple declarative language, *PML*, that allows developers to describe touch gestures as strings directly within their UI-related code (e.g., when implementing a button class). PML consists of tokens, similar to the elements of regular expressions. The two main token types are 1) area tokens and 2) transition tokens. Developers use area tokens to define the user's finger location relative to a GUI element (e.g., the area token N means "north of", e.g., a button). Moreover, they use transition tokens that chain area tokens to describe a sequence of finger locations, in other words, a touch gesture (e.g., `N->C->S` means "from north to center to south", i.e., vertically crossing, e.g., a button). Figure 10.3 shows examples.

There are further tokens, such as `O` ("origin") which denotes the area at touch down (e.g., to implement a gesture that might start anywhere). Tokens can also be stacked (e.g., `NN` is an area further north of the GUI element than `N`). In addition, tokens can be modified. Such modifiers include `d`, `m`, `u` to distinguish specific touch events (down, move, up) provided by the OS (here: Android). For example, `Cu` implements "lift on button" whereas `Cdu` means that both down and up events have to hit the button. The *ProbUI* paper lists further tokens [Buschek and Alt 2017].

10.3.3 Deriving Probabilistic Models

ProbUI 's internal algorithm takes the developers' PML statements, combined with the visual GUI properties (e.g., location and size of a button) to automatically derive probabilistic models. Figure 10.3(b) visualizes example models alongside the corresponding PML statements.

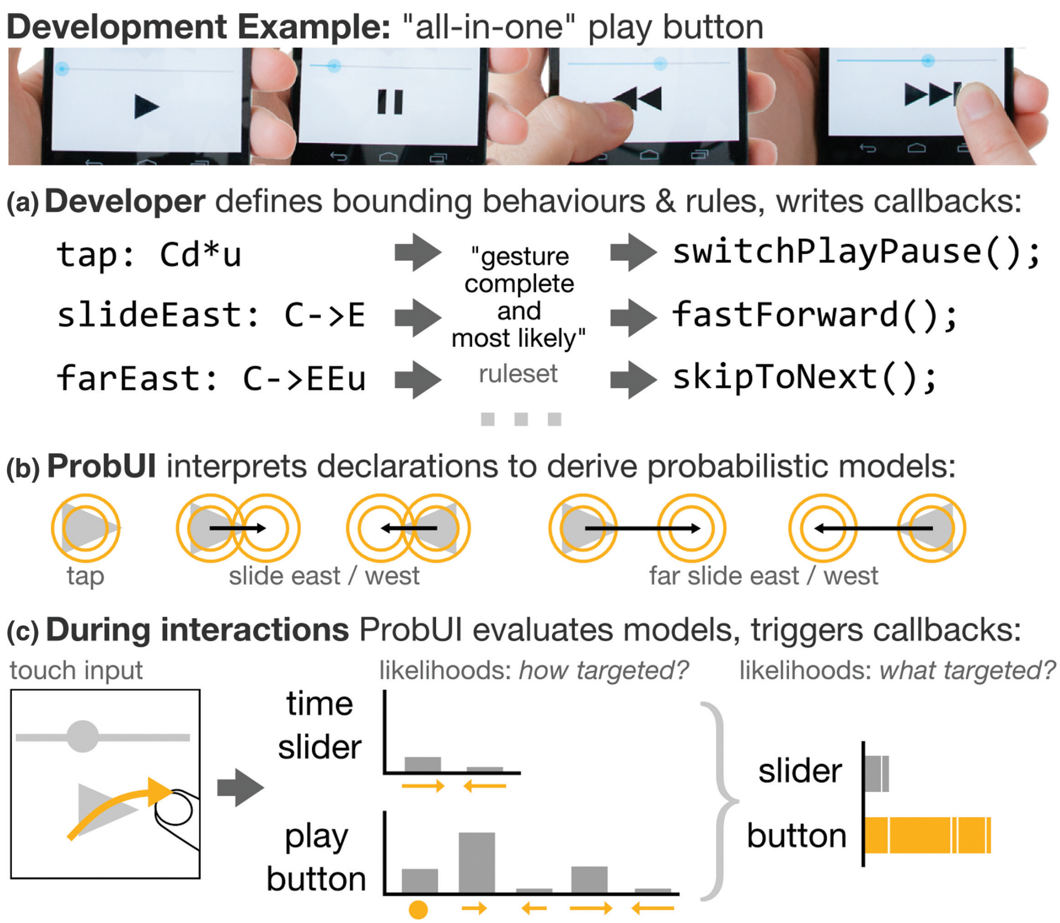

Figure 10.2 *ProbUI* development example. In this example case, we implement a novel "play" button for a music player app. The button integrates play/pause (on tap), fast back/forward (on short slide left/right), and skip to previous/next track (on long slide left/right): (a) First, the developer uses a declarative language to specify gestures for this GUI element (e.g., tap, slides). (b) *ProbUI* takes these declarations plus the GUI properties (e.g., button size, location) to automatically derive simple probabilistic gesture models. (c) During use, *ProbUI* continuously evaluates the incoming touch events to estimate the probability of each behavior, as well as the probability of each UI element. Reprinted with permission from Buschek and Alt [2017].

Formally, these models are HMMs [Rabiner 1989, Barber 2012]. HMMs are probabilistic graphical models useful for modeling sequences (e.g., here sequences of touch points x, y). They model that a sequence of observed data (e.g., touch points) results from a sequence of "hidden" states (e.g., here the areas such as `N`) whereby each such state emits observable data according to some probability distribution.

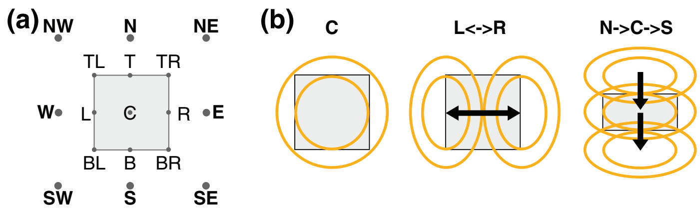

Figure 10.3 Examples for (a) area tokens (shown relative to a square button), and (b) derived models (HMMs, visualized with sigma ellipses for states, and arrows for transitions). Reprinted with permission from Buschek and Alt [2017].

These models are "Markovian" since the transition from one state to another only depends on the last state. HMMs can be used, for example, to evaluate how likely an observed sequence of data points is, or what the most likely sequences of hidden states is given the sequence of observed data. Intuitively, an HMM in this use case can be thought of as a graph of screen areas: Each area (i.e., *state* in the usual HMM terminology) is represented by a probability distribution (i.e., *emission distribution*; here: Gaussian). Multiple such states are connected by weighted transitions. For a more formal and detailed general treatment of HMMs see the related work (e.g., Rabiner [1989], Barber [2012]). Drawing a connection to Chapter 3, these HMMs can be seen as a lattice model for decoding, with states representing finger locations (touch areas) which compose a gesture trajectory, instead of words composing a sentence.

In particular, in *ProbUI* each HMM is used to evaluate the likelihood of the user's current sequence of touch points given the behavior represented by the HMM. This touch sequence processing is thus an example of processing a *discrete-time signal* (see Section 3.2.1). Moreover, compared to the Gaussian mixture models (GMMs) used for gesture recognition in Chapter 8 CS4, HMMs also model the *transitions* between the Gaussians. Thus, gestures are recognized based on both *where* the touch points occur, and also in which *order* they occur at these different locations. To create these HMMs, *ProbUI* needs to derive the following information (see e.g., Rabiner [1989], Barber [2012]):

- *States* of the HMM: *ProbUI* creates one state per area included in the PML statement. The location (i.e., mean) and "shape and size" (i.e., covariance matrix) of the Gaussian emission distribution of each state is derived by the location and size of the corresponding UI element (e.g., button location and size), as shown in Figure 10.3.
- *Transitions* of the HMM: The transition tokens (->, <->) inform the HMM's transition matrix. In brief, *ProbUI* sets positive transition probabilities for

transitions indicated by the tokens (e.g., `N->C` leads to a positive probability in the "N to C" cell of the transition matrix). The exact transition values are simply uniform defaults (and practically not too important as long as they are used consistently), but they can also be manually specified by the developer. A Laplace correction is applied such that all states are connected with at least a tiny probability. This ensures that the model can output alternative hypotheses (e.g., the user actually moves the finger in the opposite direction).

- *Starting probabilities* of the HMM's states: Finally, the starting probabilities of the HMM's states are informed by the order of the area tokens. The state corresponding to the leftmost token gets a positive starting probability (e.g., the north state in `N->C->S`). In the case of two-way gestures (e.g., rubbing left-right `L<->R`), both the first and last state get a positive starting probability. Again, a Laplace correction ensures that the model can output alternative hypotheses. Developers can easily overwrite all these probabilities with manual specifications, if desired.

These HMMs are then used internally for probabilistic inference during interaction. The overall probabilistic UI model in *ProbUI* is defined as the following factorization of the joint distribution over touch sequences t, touch behaviors b, and elements e:

$$p(t, b, e) = p(t|b)p(b|e)p(e) \tag{10.1}$$

In that model, the HMMs derived from the developers' PML statements are used to evaluate touch input to get $p(t|b)$, the probability of a touch sequence t given a behavior b. This is combined with $p(b|e)$, the probability of a behavior b given an element e (i.e., prior over possible touch gestures for a given GUI element). Finally, $p(e)$ defines the prior over elements (e.g., uniform or based on past usage).

During interaction, this models allows *ProbUI* to infer $p(b|e, t)$ (probability of behaviors per element) and $p(e|t)$ (probability of the elements). These probabilities are useful to assess what touch behavior(s) the user is most likely performing and at which GUI element(s). Figure 10.4 provides an overview of the inference process, using these HMMs. The *ProbUI* paper describes further details [Buschek and Alt 2017].

10.3.4 Rules and Event Handling

Finally, *ProbUI* also enables developers to write rules (again as strings directly in the UI-related code), using previously defined touch behaviors. For this, PML supports

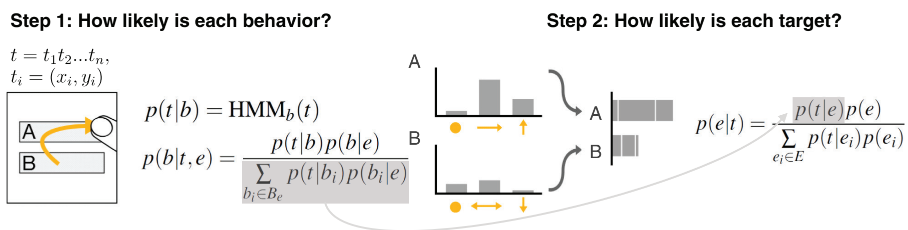

Figure 10.4 Inference process—from a sequence of touch points (x_i, y_i) to probabilities for behaviors $p(b|t, e)$ and GUI elements $p(e|t)$. From left to right: The input is a touch trajectory, that is a sequence of x, y coordinates. In step 1, the likelihood of the sequence given each behavior b is evaluated using b's HMM. With the Bayes rule, this allows us to infer $p(b|t, e)$, the distribution over behaviors per GUI element e, which is visualized in the center bar plot (elements A and B with example behaviors, i.e., the small orange arrows/dots). In step 2, the likelihoods are integrated over all behaviors per GUI element (i.e., visually: "stacking the bars" in the figure), which yields $p(t|e)$ (and this is the denominator from step 1, as indicated by the shading and arrow). With the Bayes rule, this allows us to infer $p(e|t)$, the distribution over GUI elements, given the user's current touch input. Both $p(b|t, e)$ and $p(t|e)$ can be used by developers, for example, to implement feedback and adaptations.

behavior labels (e.g., `swipe right: L->R`). These labels can then be referred to in rules, combined with keywords that relate to certain events and system states. For example, the rule `swipe_right on complete` triggers once the swipe has been completed (i.e., the finger has moved from the left to the right). Developers attach callbacks to these rules to react to such events. The examples in the following section practically illustrate this from a developer perspective. From the system's perspective, the current touch sequence is evaluated using the defined HMMs to infer the most likely state sequence (using the Viterbi algorithm [Rabiner 1989, Barber 2012]). This sequence is then matched against the sequence defined in the PML statement to see which rule keywords are fulfilled (e.g., is the gesture `complete`?) and thus which rules to trigger. Note that the underlying idea of decoding touch sequences into sequences of area tokens is related to the decoding of touch keyboard input into words in Chapter 7 CS3.

10.4 Development Examples

This section further explains the concepts and use of *ProbUI* for developers in a way similar to a tutorial. In particular, we present and discuss several examples, implementing both more "traditional" as well as novel adaptive widgets. All code examples are given in Java/Android.

10.4.1 Example 1: A Simple Counter Button

In this first example (adapted from our paper [Buschek and Alt 2017]), we implement a button that simply counts the number of times the user has vertically crossed the button with a finger swipe.

This example shows the basic components of developing with *ProbUI*. Going step-by-step through the code, we first implement a new button class (line 1), extending a basic button class provided by the framework. Then, we overwrite the `onProbSetup` method, which is called by the framework when initially creating the GUI.

Within this method, developers put their setup code, for example, to define touch behaviors and rules. In this example, we add one such behavior in line 6 (`"across: N->C->S"`), which describes a vertical slide across the button (i.e., moving the finger from an area north of the button to its center, then further down to the area south of it).

Moreover, we also add a rule in line 8 (`"across on complete"`), which uses our just defined label (`"across"`) to refer to our vertical slide. The keyphrase `"on complete"` tells *ProbUI* to check whether the swipe has just been completed. In addition, the keyphrase `"is most_likely"` checks whether this is the most likely behavior among the possible behaviors for this button (in this case there's just one anyway, but one could add, for example, another slide in a different direction). In other words, this rule evaluates to “true” at runtime if the system's inferred most likely area sequence indicates that the user has moved the finger from north of the button to its center and just entered the south, thus completing the gesture in this moment. Finally, we implement a callback method (lines 9 and 10), which is called when the rule is evaluated to “true”; in this example, the callback simply increases a counter variable.

Following this piece of code, the developer can implement any other aspects of this button class as usual. For example, we might implement visual feedback (e.g., changing the button's text to show the current counter value).

```
   public class MyButton extends ProbUIButton {
2      private int counter;
       // Called by the manager when setting up the GUI:
4      public void onProbSetup() {
           // Add a touch behaviour to this button:
6          this.core.addBehaviour("across: N->C->S");
           // Add a rule with callback:
8          this.core.addRule("across on complete and across is ...
               most_likely",
```

```
                new RuleListener() { public void onSatisfied() ...
                    {
                    counter++;
            }});
        }
        // ... rest of the class
```

Note that already in this basic example, the framework does significant useful work in the background: With this short piece of code, we end up with a probabilistic model for the defined slide (a simple HMM). Moreover, if we add multiple such buttons to an interface, *ProbUI* will automatically infer and evaluate which one to activate (and count up) in a consistent probabilistic fashion, which includes considering the full finger trajectory during the slide.

Follow-up ideas: The current button only counts downward crossings. As an exercise for the reader, this example could be extended with a second behavior, rule, and counter to detect and separately count downward and upward swipes across the button. It could be similarly extended with horizontal crossings.

10.4.2 Example 2: Swiping Through a Gallery

This second example is adapted from the developer study from our paper [Buschek and Alt 2017]. Here, we implement an image gallery view widget that reacts to swiping left/right. In particular, the two swipes are used to transition to the next/previous image in the gallery.

Looking at the code below step-by-step, we again overwrite the `onProbSetup` method, as in the previous example (rest of class not shown). We first define the two swipe behaviors in lines 4 and 5 (`"swipe_l: Od->Lu"` and `"swipe_r: Od->Ru"`). Note that this example uses the `O` token, which denotes that the gesture originates at touch down (i.e., the gesture is relative to its origin). This allows the user to perform the slide anywhere, since in this example we do not care whether slides are performed, for example, on the center of the image or at its bottom.

Next, we add two rules (lines 8–13 and 16–21): As in the first example, the rules trigger a callback on completing the most likely gesture (here: either left or right swipe). In the callback, we call either a method for loading the previous image (line 11: `prev()`) or the next one (line 19: `next()`). The details of these methods are not shown here, since they are not related to *ProbUI*.

```
   public void onProbSetup() {
2
        // Add two behaviours for swipes from the touch centre ...
            to the left/right
```

```
        this.core.addBehaviour("swipe_l: Od->Lu");
        this.core.addBehaviour("swipe_r: Od->Ru");

        // Add a rule: performing a left swipe triggers the ...
            prev() method
        this.core.addRule("prev: swipe_r on complete and swipe_r ...
            is most_likely",
            new PMLRuleListener() {
                public void onRuleSatisfied(String event, int ...
                    subsequentCalls) {
                    prev();
            }});

        // Add a rule: performing a left swipe triggers the ...
            next() method
        this.core.addRule("next: swipe_l on complete and swipe_l ...
            is most_likely",
            new PMLRuleListener() {
                public void onRuleSatisfied(String event, int ...
                    subsequentCalls) {
                    next();
            }});
    }
```

Follow-up ideas: As an exercise for the reader, this example could be extended to account for vertical swipes as well, for instance to navigate between different photo albums (e.g., swipe up for previous album, swipe down for next album). Targeting another use case, this idea could be transferred, for example, to a music player interface (e.g., swiping on an album cover for previous/next songs). It is an interesting question to think about if and how probabilities might be used in these scenarios to inform useful adaptations and visual feedback (also see the next example).

10.4.3 Example 3: Transition Effects Using *ProbUI* Probabilities

This third example extends the previous one: We now add a transition effect that blends over from one image to another, using the probabilities provided by *ProbUI*. Conceptually, we want to tie the new image's opacity to the swipe probability. In other words, the previous/next image becomes more visible with increasing confidence that the user is indeed currently performing a left/right swipe. On completing the slide, the image is then actually changed, as implemented in the previous example.

To achieve this, we overwrite the widget's drawing function (called by *ProbUI*, tied to the usual Android UI drawing system). As the code snippet below shows, developers can easily access the current probability of user behavior at any point in their code (lines 4 and 5). These values can be used freely; here, we use them to 1) decide if we should blend over to the next or the previous image (if-else statement, lines 11 and 17), and to 2) set the opacity of the previous/next image (lines 12 and 18). The other lines simply call drawing commands provided by the standard Android API.

It is worth pointing out here that the method calls `getBehaviourProb("swipe_l")` (line 4) and `getBehaviourProb("swipe_r")` (line 5) yield the current probabilities of performing the swipes, *regardless* of if (or how far) these slides have been completed by the user. Completion can be checked via rules (see previous example). This way, probabilities of behaviors enable developers to track user intention "live" —in this example to show feedforward (i.e., showing the user which image would be made fully visible if the swipe is completed).

```
   public void drawSpecific(Canvas canvas) {
2
       // Get the behaviour probabilities for both swipes:
4      double probSwipeLeft = ...
           this.core.getBehaviourProb("swipe_l");
       double probSwipeRight = ...
           this.core.getBehaviourProb("swipe_r");
6
       /* Tie the opacity of the previous/next image to the ...
           probability
8         of the right/left swipe, respectively. */

10     // if swipe left more likely and there is another image:
       if (probSwipeLeft > probSwipeRight && ...
           this.previewImageNext != null) {
12         this.previewPaint.setAlpha((int) (probSwipeLeft * 255));
           canvas.drawBitmap(this.previewImageNext, null,
14             this.canvasRect, this.previewPaint);
       }
16     // else (swipe right more likely) and there is a ...
           previous image:
       else if (this.previewImagePrev != null) {
18         this.previewPaint.setAlpha((int) (probSwipeRight * ...
               255));
           canvas.drawBitmap(this.previewImagePrev, null,
20             this.canvasRect, this.previewPaint);
       }
22 }
```

Follow-up ideas: This example could be extended with many different kinds of visual feedforward (e.g., what could be tied to the probabilities instead of transparency?). In this gallery use case this might result in different transition effects. Moreover, the simple transparency effect (or other effects) could be implemented for other widgets to signal activation or completion of actions (e.g., for sliders, selection of list/grid items, selection of map points/objects, etc.). It might be interesting to explore the impact on user experience if this is developed further as a UI concept where this kind of feedforward is the norm. Inversely, one could think about using the *lowest* probabilities, for example, to give the user visual cues on *what else* could be done with a widget in contrast to how the user is currently using it (e.g., to improve discoverability of gesture-enabled functionalities, cf. Bau and Mackay [2008]).

10.4.4 Further Examples: UI Elements that Adapt to Hand Postures

The previous examples have demonstrated fundamental aspects of using the framework. Equipped with these basics, we can now turn towards cases that more clearly show the value of the framework for building *adaptive* interface elements. Figure 10.5 shows three such widgets:

In Figure 10.5(a) and (b), an adaptive slider bends itself to match the thumb's reach and movement arc, whereas Figure 10.5(c) shows the sliders' feedforward in cases of uncertain user input (e.g., thumb moving in a yet unclear trajectory in between two sliders. In Figures 10.5(d) and (e), an adaptive menu button is either opened in a straight line on tap, or in an arced layout on a flick with the finger/thumb; the latter enables users to reach the top menu items during one-handed use, even on devices with a larger screen. Finally, Figure 10.5(f) shows an adaptive contact list that swaps the alignment of contact portrait and buttons based on the scrolling trajectory, such that the user's thumb is always close to the buttons and never occludes the contact portrait.

The visual adaptation of these widgets is more sophisticated than the preceding examples. Implementing, say, the bending animations for sliders and menu item layouts is not a part of *ProbUI*, but rather relies on the Android API. However, *ProbUI* makes it easy to *trigger* and *manage* these visual changes, for example, to keep track of what should be displayed at which point during an interaction, and to reach decisions related to such feedback/feedforward.

Overall, these widgets are implemented in the same way as the previous examples: Developers set up multiple behaviors and use rules with callbacks to react to them. For example, the bending slider has five behaviors, one for each bending direction, plus one for the default straight state (see Buschek and Alt [2017]).

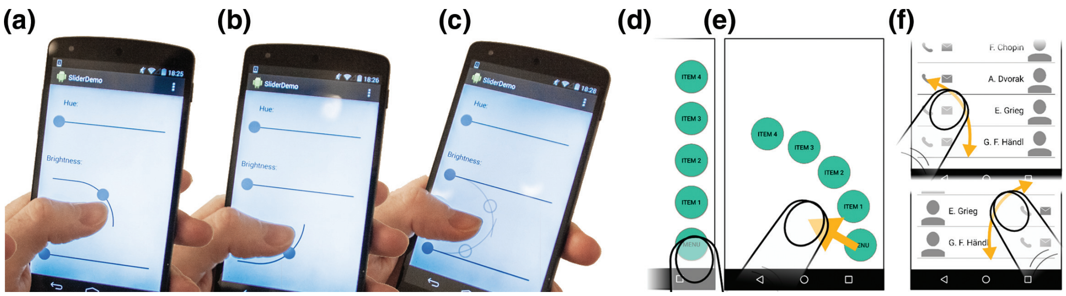

Figure 10.5 Example UI elements that adapt to the user's hand posture and finger in use: bending sliders, adaptive menu item layouting, and adaptive alignment of portraits and buttons in a contact list. See text for further details. Reprinted with permission from Buschek and Alt [2017].

Similarly, the adaptive contact list has the following three behaviors and two related rules:

```
this.core.addBehaviour("straight: T<->B");
this.core.addBehaviour("arc_left: L<->B");
this.core.addBehaviour("arc_right: R<->B");

this.core.addRule("arc_right is complete and arc_right is ...
    most_likely",
    new PMLRuleListener() {
        public void onRuleSatisfied(String event, int ...
            subsequentCalls) {
            updateAlignment(ALIGN_LEFT);
    }});

this.core.addRule("arc_left is complete and arc_left is ...
    most_likely",
    new PMLRuleListener() {
        public void onRuleSatisfied(String event, int ...
            subsequentCalls) {
            updateAlignment(ALIGN_RIGHT);
    }});
```

Note that the `"straight"` behavior is not linked to a rule and callback, since there are only two alignments of the list entry GUI elements (see Figure 10.5[f]: portraits left, buttons right—or *vice versa*). However, by adding this straight behavior, none of the two arced behaviors becomes the most likely one when the user is actually scrolling in a straight trajectory (e.g., with the index finger). This stops the list from flipping its alignment for straight scrolling trajectories. Thus, this example demonstrates how developers can make use of the probabilistic reasoning framework to "catch" user behavior that should *not* be used for adaptation. Supporting this aspect is crucial for facilitating the development of *robust* adaptations, since unwanted adaptation (or "over-adaptation") and resulting unpredictability are well-recognized problems of adaptive UIs (e.g., see Browne et al. [1990], Gajos et al. [2008], Lavie and Meyer [2010]).

10.5 Reflection from a Developer Perspective

The preceding section demonstrated *ProbUI*'s use in the development of several example interface widgets. In this section, we reflect on *ProbUI*'s concepts with regard to interactive system developers. Besides reflections on a conceptual level, this section is based on empirical results from an online survey ($N = 33$, 11 female, mean age 25 years) and a workshop ($N = 8$, 3 female, mean age 25 years), both conducted with Android developers (also see Buschek and Alt [2017]).

In the workshop, we introduced *ProbUI* in a short 20-min presentation, plus Q&A. We then provided six short coding projects in which developers had to implement widgets (similar to the examples in this chapter) using *ProbUI*. We encouraged questions during the workshop and recorded developers' "thinking aloud." The workshop concluded with a questionnaire and interview.

10.5.1 Declarative Language (PML)

One key aspect of our framework for developers is our declarative language, PML, which developers use to define touch behaviors. PML was first addressed in the online survey: Here, our goal was to evaluate how easy it is to learn and understand PML, and to gather first feedback on the concept. The survey explained PML similar to an online tutorial. Afterwards, it asked developers to 1) translate gestures given as videos/images into PML, to 2) select a gesture based on a given PML statement, and finally to 3) write PML statements themselves for a given gesture.

The participating developers took an average of 8:06 min to read the explanation and complete all tasks with a score of 95.45% correct answers. In the Likert questions, 97% rated it as easy to understand, 91% as easy to write. Overall, 75% were interested in actually employing it in app development. Together, these results indicate that developers learned to read and use PML quickly, with a positive attitude towards the concept.

Further insights into the concept of using a declarative language for setting up probabilistic gesture models result from our developer workshop. Here, we found that PML is easy to use after an initial introduction, matching the results from the online survey. However, during actual use, we observed a learning curve: For example, a common question raised by participants addressed the two sets of area tokens (around GUI element: N,E,S,W vs. on GUI element: T,R,B,L; see Figure 10.3[a]). It was unclear to the developers when to use which set of tokens. In fact, *ProbUI* is very lenient about, for example, swapping north/top (N/T), since input is evaluated probabilistically. Reflecting on the concept, it might thus be useful to improve the tokens' explanation, or even think about redesigning PML to clearly favor one set of area tokens and present using both as a more advanced option for refined setups.

The workshop also revealed that developers would like to receive IDE support for the declarative language. For example, while modern IDEs have strong capabilities to auto-complete code, they do not know about PMLs tokens and syntax and thus cannot support syntax-checking or auto-completion for PML statements. This could be fixed, for example, with an IDE plugin. Following one developer's suggestion, we could also provide string constants for PMLs keyphrases, which would (partly) enable auto-completion in unmodified IDEs.

10.5.2 Probabilistic GUI Concept

Besides PML, *ProbUI* provides a probabilistic GUI concept that facilitates development of adaptive UIs. The examples in Section 10.4 demonstrate how these probabilities might be used, for example, to implement visual feedback. Working with the option of utilizing probabilistic information anywhere in UI code is a key aspect of development as enabled by *ProbUI*. Thus, we reflect on the related findings from our developer workshop here.

Overall, working with probabilities was unfamiliar for the developers participating in our workshop. Nevertheless, they welcomed the idea and practical value, as evident, for example, from many suggestions on how they might use such probabilities for interfaces beyond those included in the workshop tasks. As with PML, we observed a learning process: Several initial key questions repeatedly occurred in the workshop sessions. These are particularly insightful to reflect on.

First, a common question addressed the distinction between the two types of probabilities provided by *ProbUI*; that is, 1) the probabilities of touch behaviors (e.g., swipe left) versus 2) the probabilities of GUI elements (e.g., button X). This distinction and the practical value of both types became clearer over the course of the tasks. One idea to better support learning about this distinction is an object-oriented access method, suggested by one of the participants and now implemented in *ProbUI*: In addition to getter methods for probabilities provided by the "core" *ProbUI* system object (e.g., `this.core.getBehaviourProb("swipe\_l")`), the API now also lets developers first use a getter for either the GUI element or a specific behavior—and only then call a getter for a probability on that object (e.g., `this.getBehaviour("swipe_l").getProbability()`).

This API change ties in with the second key question revealed by our workshop, which addressed object-oriented development. Many methods in *ProbUI* use string identifiers, since those also appear as part of the declarative language PML (e.g., the behavior labels as in `"swipe_left: R->L"`). While PML was well-received, developers also expressed the wish to work with the resulting components in a typical object-oriented way. As a result, *ProbUI* now returns behavior objects when defining behaviors. Developers can use these objects to access probabilities, manually refine the behavior parameters, and so on. Going a step further, we might also support an object-based declaration instead of PML altogether, also for the rules. Comparing this against the declaration-as-strings API approach presents an interesting aspect for a future study with developers. Overall, however, *ProbUI* could also support both API approaches in parallel, since the framework internally already uses objects to represent all components anyway.

10.5.3 Generalizing GUI Target Representations

In traditional GUI frameworks for mobile apps and also for websites, GUI target areas (e.g., active area of a button) are described as rectangles (often called "bounding boxes"). Boxes are a simple representation for GUI targets which implies three limitations for developers:

1. Bounding boxes are a *discrete* representation; they cannot deal with uncertain input, since a touch point is either in or out of a box. Thus, there is no inherent notion of uncertainty.
2. There is a *one-to-one mapping* from boxes to GUI targets. For example, each button has exactly one bounding box that describes where/how expected user input for this button takes place.
3. Bounding boxes are a *static* representation. They only describe simple tapping well. In contrast, many other touch behaviors are dynamic and thus not adequately represented by boxes. For example, users might slide [Yatani et al. 2008, Moscovich 2009], rub [Roudaut et al. 2009], cross [Apitz et al. 2008, Perin et al. 2015], and encircle GUI targets [Choe et al. 2009, Ka 2013].

Reflecting on the concepts of *ProbUI*, we see that it generalizes GUI target representations from bounding boxes to what we call "bounding behaviors." In particular, this generalization addresses the three limitations listed:

First, our bounding behaviors are *probabilistic* representations (HMMs). This supports providing the various benefits motivated throughout this chapter, such as inferring user intention, giving continuous feedback/feedforward, and robustly adapting the GUI.

Second, our framework allows developers to attach *more than one* bounding behavior to each GUI element. This enables GUIs to anticipate and address variations in user behavior (e.g., due to different hand postures) or entirely different ways of using one GUI element (e.g., target selection by tapping vs. crossing). This supports not only different user preferences for interaction styles but may also better account for the skills of specific user groups (e.g., motor impairments) and contexts (e.g., stationary use at home vs. less precise mobile use).

Finally, by using HMMs, our bounding behaviors better represent *dynamic* user behavior which unfolds over time (i.e., here: touch gestures), compared to the static bounding boxes. As our examples show, this enables developers to realize and work with a consistent probabilistic representation of many (previously non-probabilistic) touch input behaviors proposed in the HCI literature (cf. examples in Buschek and Alt [2017]).

10.5.4 Limitations and Extensions

We further reflect on the concepts of the framework, beyond the survey and workshop, based on discussions with fellow researchers and practitioners. Here, it is also insightful to address the main limitations of *ProbUI* in its current state:

One constraint is set by the choice for the declarative language: Complex gestures are much more difficult to express with a sequence of tokens than with demonstration. One might argue that gestures tied to GUI elements should be limited in their complexity anyway for reasons of usability. Nevertheless, it would clearly be useful to support multitouch gestures more directly. *ProbUI* already keeps track of touch events by all fingers and includes simple declarative statements addressing multiple fingers (e.g., number of fingers used to define a two-finger slide). A useful extension of this could, for example, allow developers to index area tokens with finger IDs. A radically different approach to integrating complex gestures could employ a ML gesture recognizer instead of declarations and feed the resulting probabilities to the remaining parts of the framework (cf. related work in Section 10.2.3).

Regarding the probabilistic "backend" of the framework, it should be noted that the presented concept does not necessarily result in models that fit the users' actual behavior well in the ML sense. Since the models are only informed by declarations and GUI properties (layout, sizes)—and not from past user behavior data—the HMMs should be seen as rather rough approximations of behavior. Nevertheless, they provide a *systematic* probabilistic treatment of input behavior.

ProbUI enables developers to access probabilities about behaviors and GUI targets anywhere in the UI code. However, the other direction is relevant as well, that is, accessing GUI information in the probabilistic reasoning process.

ProbUI currently supports this to a limited extent: Our implemented mediator class accesses properties such as visibility and "enabled" states (to ignore invisible/disabled GUI elements for reasoning). It also keeps track of the GUI elements' locations (e.g., while scrolling) to update the behavior models accordingly (i.e., move the HMMs' emission states locations). However, the framework currently does not directly support reasoning with larger GUI "states" (e.g., considering the user's recent navigation history in the app). This is an opportunity for future extensions. Nevertheless, developers can already use the probabilities provided by *ProbUI* to write their own systems that use such information (e.g., implementing a state machine with transitions based on behavior probabilities). A very simple example of this is given by our adaptive widgets (e.g., the slider has five bending "states").

Finally, the probabilistic reasoning requires additional computations, compared to a traditional GUI framework. For each touch event, *ProbUI* delegates the event data to all GUI elements, evaluates their HMMs, checks the defined rules, runs the mediator, plus other "bookkeeping" work. In our implementation, this did not incur noticeably delays (on a Nexus 5 phone) with a reasonable number of GUI elements and attached behaviors and rules. However, computational costs could become an issue for GUIs with many elements each with multiple behaviors and rules. Developers can address this only to a limited extent. To improve the framework in this regard, the evaluation of behaviors and so on can be parallelized: Conceptually, this is easily possible since the evaluation of one HMM is independent of the others.

10.6 Conclusion and Outlook

This chapter presented *ProbUI* as a case study of a framework for developers of intelligent and adaptive UIs. In particular, *ProbUI* addressed mobile touch interfaces. As a key insight for supporting their development, it combines the ease-of-use of declarative specification of user behavior (here: touch gestures) with the benefits of probabilistic modeling and reasoning during interaction.

We demonstrated the use of the framework from a developer perspective through several coding examples, followed by a reflection on the underlying concepts, based on a survey and workshop with Android developers. We conclude that declarations are an adequate and useful approach to support developers in specifying expected user input behavior (and its variations) directly embedded into UI code. Furthermore, the novel automatic probabilistic "backend" of our framework enables developers to work with probabilities without coding the underlying models (here: HMMs) by hand. In particular, developers can access and use probabilistic information about current user behavior regarding both likely input/gestures and likely GUI targets. These probabilities are kept up-to-date automatically and continuously during unfolding user interactions. Finally, our concepts generalize GUI target representations from bounding boxes to "bounding behaviors," which better account for dynamic and uncertain user input behavior, and also allow developers to anticipate and account for multiple different ways in which users might want to interact with a given GUI element.

Overall, *ProbUI* thus presents a case study of how we can combine previously disparate concepts (such as declaration and probabilistic reasoning) at the intersection of ML and HCI to facilitate their integration into novel interfaces and to practically support the developers of these future interactive systems.

Code and additional material are available at: http://www.medien.ifi.lmu.de/probui/.

10.7 Follow-up Questions

Here are some further ideas for following up on the contents of this chapter after reading:

- UI and interaction design—This chapter and the *ProbUI* paper list several example widgets. Can you come up with further widgets that benefit from probabilistic input handling? How could you show that there is a practical benefit for the user?
- Practical—Try out the framework for yourself: Download *ProbUI* and try out the examples, the follow-up ideas described after each example in this chapter, or your own ideas from the previous question. Do you encounter practical limitations when realizing your ideas? If so, what would need to change conceptually?
- Conceptual—How could the ideas of *ProbUI* be adapted or extended for input beyond a touchscreen? For example, think about augmented reality (AR)/virtual reality (VR) applications with typical controls like mid-air gestures—that is, three-dimensional (3D) gestures instead of two-dimensional (2D) gestures. What would probably need to be changed and what could be reused from 2D?

10.8 Further Reading

This list provides pointers to related work and further reading on the topics of this chapter:

- More on *ProbUI* itself can be found in the Conference on Human Factors in Computing Systems (CHI'17) paper on this framework [Buschek and Alt 2017].

- For other frameworks for probabilistic GUIs, including concepts for treating GUI states in a probabilistic fashion, see the work by Schwarz et al. [2010, 2011, 2015].
- The *Proton* papers give a detailed treatment of using a declarative language for specifying touch gestures [Kin et al. 2012a, 2012b].
- For further motivation and use of adaptive UIs for diverse user groups, see for example, this overview on "Ability-based Design" [Wobbrock et al. 2018].
- Further details and ideas for probabilistic models of mobile finger touch input can be found, for example, in these papers [Bi et al. 2013, Bi and Zhai 2013, Yin et al. 2013, Weir et al. 2014].
- For more background on touch input, see for example, these studies by Holz and Baudisch [2010, 2011].
- A broader view on modeling user behavior in a computational perspective on HCI can be found in Oulasvirta et al. [2018].
- A general textbook on probabilistic modeling, for example, is Barber[2012].

References

G. Apitz, F. Guimbretière, and S. Zhai. May 2008. Foundations for designing and evaluating user interfaces based on the crossing paradigm. *ACM Trans. Comput.-Hum. Interact.* 17, 2, 9:1–9:42. ISSN: 1073-0516. DOI: http://doi.acm.org/10.1145/1746259.1746263.

D. Barber. 2012. *Bayesian Reasoning and Machine Learning*. Cambridge University Press. http://web4.cs.ucl.ac.uk/staff/D.Barber/textbook/090310.pdf.

O. Bau and W. E. Mackay. 2008. OctoPocus: A dynamic guide for learning gesture-based command sets. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology, UIST'08*. ACM, New York, NY, 37–46. ISBN: 978-1-59593-975-3. DOI: http://doi.acm.org/10.1145/1449715.1449724.

X. Bi and S. Zhai. 2013. Bayesian touch: A statistical criterion of target selection with finger touch. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology, UIST'13*. ACM, New York, NY, 51–60. ISBN: 978-1-4503-2268-3. DOI: http://doi.acm.org/10.1145/2501988.2502058.

X. Bi, Y. Li, and S. Zhai. 2013. Fitts law: Modeling finger touch with Fitts' law. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'13*. ACM, New York, NY, 1363–1372. ISBN: 978-1-4503-1899-0. DOI: http://doi.acm.org/10.1145/2470654.2466180.

D. Browne, P. Totterdell, and M. Norman (Eds.). 1990. *Adaptive User Interfaces*. Academic Press, San Diego, CA.

D. Buschek and F. Alt. 2015. TouchML: A machine learning toolkit for modelling spatial touch targeting behaviour. In *Proceedings of the 20th International Conference on*

Intelligent User Interfaces, IUI'15. ACM, New York, NY, 110–114. ISBN: 978-1-4503-3306-1. DOI: http://doi.acm.org/10.1145/2678025.2701381.

D. Buschek and F. Alt. 2017. ProbUI: Generalising touch target representations to enable declarative gesture definition for probabilistic GUIs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'17*. ACM, New York, NY, 4640–4653. ISBN: 978-1-4503-4655-9. DOI: http://doi.acm.org/10.1145/3025453.3025502.

D. Buschek, O. Schoenleben, and A. Oulasvirta. 2014. Improving accuracy in back-of-device multitouch typing: A clustering-based approach to keyboard updating. In *Proceedings of the 19th International Conference on Intelligent User Interfaces, IUI'14*. ACM, New York, NY, 57–66. ISBN: 978-1-4503-2184-6. DOI: http://doi.acm.org/10.1145/2557500.2557501.

G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. 2003. A unifying reference framework for multi-target user interfaces. *Interact. Comput.*, 15, 3, 289–308. DOI: https://doi.org/10.1016/S0953-5438(03)00010-9.

E. K. Choe, K. Shinohara, P. K. Chilana, M. Dixon, and J. O. Wobbrock. 2009. Exploring the design of accessible goal crossing desktop widgets. In *CHI'09 Extended Abstracts on Human Factors in Computing Systems, CHI EA'09*. ACM, New York, NY, 3733–3738. ISBN: 978-1-60558-247-4. DOI: http://doi.acm.org/10.1145/1520340.1520563.

L. Findlater and J. Wobbrock. 2012. Personalized input: Improving ten-finger touchscreen typing through automatic adaptation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'12*. ACM, New York, NY, 815–824. ISBN: 978-1-4503-1015-4. DOI: http://doi.acm.org/10.1145/2207676.2208520.

K. Z. Gajos, K. Everitt, D. S. Tan, M. Czerwinski, and D. S. Weld. 2008. Predictability and accuracy in adaptive user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'08*. ACM, New York, NY, 1271–1274. ISBN: 978-1-60558-011-1. DOI: http://doi.acm.org/10.1145/1357054.1357252.

M. Goel, L. Findlater, and J. Wobbrock. 2012. WalkType: Using accelerometer data to accomodate situational impairments in mobile touch screen text entry. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'12*. ACM, New York, NY, 2687–2696. ISBN: 978-1-4503-1015-4. DOI: http://doi.acm.org/10.1145/2207676.2208662.

M. Goel, A. Jansen, T. Mandel, S. N. Patel, and J. O. Wobbrock. 2013. ContextType: Using hand posture information to improve mobile touch screen text entry. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'13*. ACM, New York, NY, 2795–2798. ISBN: 978-1-4503-1899-0. DOI: http://doi.acm.org/10.1145/2470654.2481386.

C. Holz and P. Baudisch. 2010. The generalized perceived input point model and how to double touch accuracy by extracting fingerprints. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'10*. ACM, New York, NY, 581–590. ISBN: 978-1-60558-929-9. DOI: http://doi.acm.org/10.1145/1753326.1753413.

C. Holz and P. Baudisch. 2011. Understanding touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'11*. ACM, New York, NY, 2501–2510. ISBN: 978-1-4503-0228-9. DOI: http://doi.acm.org/10.1145/1978942.1979308.

E. Horvitz. 1999. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'99*. ACM, New York, NY, 159–166. ISBN: 0-201-48559-1. DOI: http://doi.acm.org/10.1145/302979.303030.

H. W. Ka. 2013. *Circling Interface: An Alternative Interaction Method for On-Screen Object Manipulation*. PhD thesis, University of Pittsburgh. http://d-scholarship.pitt.edu/19305/.

S. H. Khandkar and F. Maurer. 2010. A domain specific language to define gestures for multi-touch applications. In *Proceedings of the 10th Workshop on Domain-Specific Modeling, DSM'10*. ACM, New York, NY, 2:1–2:6. ISBN: 978-1-4503-0549-5. DOI: http://doi.acm.org/10.1145/2060329.2060339.

K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. 2012a. Proton: Multitouch gestures as regular expressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'12*. ACM, New York, NY, 2885–2894. ISBN: 978-1-4503-1015-4. DOI: http://doi.acm.org/10.1145/2207676.2208694.

K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. 2012b. Proton++: A customizable declarative multitouch framework. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology, UIST'12*. ACM, New York, NY, 477–486. ISBN: 978-1-4503-1580-7. DOI: http://doi.acm.org/10.1145/2380116.2380176.

T. Lavie and J. Meyer. 2010. Benefits and costs of adaptive user interfaces. *Int. J. Hum. Comput. Stud.* 68, 8, 508–524. ISSN: 1071-5819. http://www.sciencedirect.com/science/article/pii/S1071581910000145. DOI: https://doi.org/10.1016/j.ijhcs.2010.01.004. Measuring the Impact of Personalization and Recommendation on User Behaviour.

Y. Li, H. Lu, and H. Zhang. Aug. 2014. Optimistic programming of touch interaction. *ACM Trans. Comput. Hum. Interact.*, 21, 4, 24:1–24:24. ISSN: 1073-0516. DOI: http://doi.acm.org/10.1145/2631914.

H. Lü and Y. Li. 2012. Gesture coder: A tool for programming multi-touch gestures by demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'12*. ACM, New York, NY, 2875–2884. ISBN: 978-1-4503-1015-4. DOI: http://doi.acm.org/10.1145/2207676.2208693.

H. Lü and Y. Li. 2013. Gesture studio: Authoring multi-touch interactions through demonstration and declaration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'13*. ACM, New York, NY, 257–266. ISBN: 978-1-4503-1899-0. DOI: http://doi.acm.org/10.1145/2470654.2470690.

H. Lü, J. A. Fogarty, and Y. Li. 2014. Gesture script: Recognizing gestures and their structure using rendering scripts and interactively trained parts. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems, CHI'14*. ACM, New York, NY, 1685–1694. ISBN: 978-1-4503-2473-1. DOI: http://doi.acm.org/10.1145/2556288.2557263.

J. Mankoff, S. E. Hudson, and G. D. Abowd. 2000a. Interaction techniques for ambiguity resolution in recognition-based interfaces. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology, UIST'00*. ACM, New York, NY, 11–20. ISBN: 1-58113-212-3. DOI: http://doi.acm.org/10.1145/354401.354407.

J. Mankoff, S. E. Hudson, and G. D. Abowd. 2000b. Providing integrated toolkit-level support for ambiguity in recognition-based interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'00*. ACM, New York, NY, 368–375. ISBN: 1-58113-216-6. DOI: http://doi.acm.org/10.1145/332040.332459.10.1145/332040.332459.

T. Minka, J. Winn, J. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. 2014. *Infer.NET 2.6*. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

T. Moscovich. 2009. Contact area interaction with sliding widgets. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology, UIST'09*. ACM, New York, NY, 13–22. ISBN: 978-1-60558-745-5. DOI: http://doi.acm.org/10.1145/1622176.1622181.

M. E. Mott, R.-D. Vatavu, S. K. Kane, and J. O. Wobbrock. 2016. Smart touch: Improving touch accuracy for people with motor impairments with template matching. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, CHI'16*. ACM, New York, NY, 1934–1946. ISBN: 978-1-4503-3362-7. DOI: http://doi.acm.org/10.1145/2858036.2858390.

J. Musić and R. Murray-Smith. 2016. Nomadic input on mobile devices: The influence of touch input technique and walking speed on performance and offset modeling. *Hum. Comput. Interact.* 31, 5, 420–471. DOI: https://doi.org/10.1080/07370024.2015.1071195.

A. Oulasvirta, P. O. Kristensson, X. Bi, and A. Howes (Eds.). 2018. *Computational Interaction*. Oxford University Press.

C. Perin, P. Dragicevic, and J.-D. Fekete. 2015. Crossets: Manipulating multiple sliders by crossing. In *Proceedings of the 41st Graphics Interface Conference, GI'15*. Canadian Information Processing Society, Toronto, Ont., Canada, 233–240. ISBN: 978-0-9947868-0-7. http://dl.acm.org/citation.cfm?id=2788890.2788931. DOI: https://doi.org/10.1145/2670444.2670449.

L. Rabiner. Feb. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. IEEE* 77, 2, 257–286. ISSN: 0018-9219. DOI: https://dl.acm.org/doi/10.5555/108235.108253.

A. Roudaut, E. Lecolinet, and Y. Guiard. 2009. MicroRolls: Expanding touch-screen input vocabulary by distinguishing rolls vs. slides of the thumb. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'09*. ACM, New York, NY, 927–936. ISBN: 978-1-60558-246-7. DOI: http://doi.acm.org/10.1145/1518701.1518843.

Z. Sarsenbayeva, N. van Berkel, C. Luo, V. Kostakos, and J. Goncalves. 2017. Challenges of situational impairments during interaction with mobile devices. In *29th Australian Conference on Human-Computer Interaction (OzCHI'17)*. ISBN: 1234567245. DOI: https://doi.org/10.1145/3152771.3156161.

C. Scholliers, L. Hoste, B. Signer, and W. De Meuter. 2011. Midas: A declarative multi-touch interaction framework. In *Proceedings of the Fifth International Conference on Tangible, Embedded, and Embodied Interaction, TEI'11*. ACM, New York, NY, 49–56. ISBN: 978-1-4503-0478-8. DOI: http://doi.acm.org/10.1145/1935701.1935712.

J. Schwarz, S. Hudson, J. Mankoff, and A. D. Wilson. 2010. A framework for robust and flexible handling of inputs with uncertainty. In *Proceedings of the 23rd Annual ACM Symposium on User Interface Software and Technology, UIST'10*. ACM, New York, NY, 47–56. ISBN: 978-1-4503-0271-5. DOI: http://doi.acm.org/10.1145/1866029.1866039.

J. Schwarz, J. Mankoff, and S. Hudson. 2011. Monte Carlo methods for managing interactive state, action and feedback under uncertainty. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, UIST'11*. ACM, New York, NY, 235–244. ISBN: 978-1-4503-0716-1. DOI: http://doi.acm.org/10.1145/2047196.2047227.

J. Schwarz, J. Mankoff, and S. E. Hudson. 2015. An architecture for generating interactive feedback in probabilistic user interfaces. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI'15*. ACM, New York, NY, 2545–2554. ISBN: 978-1-4503-3145-6. DOI: http://doi.acm.org/10.1145/2702123.2702228.

W. Wahlster and M. Maybury. 1998. An introduction to intelligent user interfaces. In *RUIU*. Morgan Kaufmann, San Francisco, 1–13.

D. Weir, S. Rogers, R. Murray-Smith, and M. Löchtefeld. 2012. A user-specific machine learning approach for improving touch accuracy on mobile devices. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology, UIST'12*. ACM, New York, NY, 465–476. ISBN: 978-1-4503-1580-7. DOI: http://doi.acm.org/10.1145/2380116.2380175.

D. Weir, H. Pohl, S. Rogers, K. Vertanen, and P. O. Kristensson. 2014. Uncertain text entry on mobile devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'14*. ACM, New York, NY, 2307–2316. ISBN: 978-1-4503-2473-1. DOI: http://doi.acm.org/10.1145/2556288.2557412.

J. Williamson. 2006. *Continuous Uncertain Interaction*. PhD thesis, University of Glasgow.

J. O. Wobbrock, K. Z. Gajos, S. K. Kane, and G. C. Vanderheiden. May 2018. Ability-based design. *Commun. ACM* 61, 6, 62–71. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/3148051.

K. Yatani, K. Partridge, M. Bern, and M. W. Newman. 2008. Escape: A target selection technique using visually-cued gestures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'08*. ACM, New York, NY, 285–294. ISBN: 978-1-60558-011-1. DOI: http://doi.acm.org/10.1145/1357054.1357104.

Y. Yin, T. Y. Ouyang, K. Partridge, and S. Zhai. 2013. Making touchscreen keyboards adaptive to keys, hand postures, and individuals: A hierarchical spatial backoff model approach. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI'13*. ACM, New York, NY, 2775–2784. ISBN: 978-1-4503-1899-0. DOI: http://doi.acm.org/10.1145/2470654.2481384.