Chapter 8

Getting Input from Users: Forms and Controls

Sooner or later, the software you design will probably ask the user to answer some kind of question. It might even happen in the first few minutes of interaction. Where should this software be installed? What's your login name? What words do you want to search for?

These kinds of interactions are among the easiest to design. Everyone knows how to use text fields, checkboxes, and combo boxes. These input controls are often the first interface elements that novice designers use as they build their first GUIs and websites.

Still, it doesn't take much to set up a potentially awkward interaction. Here's another sample question: for what location do you want a weather report? The user might wonder, do I specify a location by city, country, postal code, or what? Are abbreviations OK? What if I misspell it? What if I ask for a city it doesn't know about? Isn't there a map to choose from? And why can't it remember the location I gave it yesterday, anyhow?

This chapter discusses ways to smooth out these problems. The patterns, techniques, and controls described here apply mostly to form design—a *form* being simply a series of question/answer pairs. However, they will also be useful in other contexts, such as for single controls on web pages or on application toolbars. Input design and form design are core skills for interaction designers, as you can use them in every genre and on every platform.

The Basics of Form Design

First, a few principles to remember when doing input and form design:

Make sure the user understands what's asked for, and why
: This is entirely context-dependent, and any generalizations here risk sounding vapid, but let's try anyway. You should write labels with words that your target users understand—plain language for novice or occasional users, and carefully chosen jargon or specialized vocabulary for domain experts. If you design a long or tedious form, explain why you need all the information you're asking for, break it up into descriptive Titled Sections, and give some reassurance that sensitive information will

be handled with care. If you're putting a button on a toolbar (or somewhere else that's too crowded for a label) and the button's function isn't immediately obvious, use a descriptive tool tip or other type of context-sensitive help to tell the user what it does.

If you can, avoid asking the question at all

Asking the user to answer a question, especially if it comes in the middle of some other task, is a bit of an imposition. You might be asking him to break his train of thought and deal with something he hadn't expected to think about. Even in the best of cases, typing into text fields isn't most people's idea of a fun time. Can you "prefill" an input control with already-known or guessable information, as the Autocompletion pattern recommends? Can you offer Good Defaults that remove the burdens of choice from most of your users? Can you avoid asking for the information altogether?

There's one glaring exception to this principle: security. Sometimes we use input controls in a challenge/response context, such as asking for passwords or credit card numbers. You obviously don't want to circumvent these security mechanisms by casually prefilling sensitive information.

Knowledge "in the world" is often more accurate than knowledge "in the head"

You can't expect human beings to recall lists of things perfectly. If you ask users to make a choice from a prescribed set of items, try to make that list available to them so that they can read over it. Drop downs, combo boxes, lists, and other such controls put all the choices out there for the user to review.

(Obviously, a user can remember his name, birthday, address, state or country, phone number, and other common personal information—and he can type such information very easily and accurately. There's no need for "knowledge in the world" in these cases; text fields work just fine, and are easier to use than drop downs.)

Similarly, if you ask for input that needs to be formatted in a specific way, you might want to offer the user clues about how to format it. Even if the user has used your UI before, he may not remember what's required—a gentle reminder may be welcome. Good Defaults, Structured Format, and Input Hints all serve this purpose. Autocompletion goes a step further by telling the user what input is valid, or by reminding the user what he entered some previous time.

Respond sensitively to errors, and be forgiving when possible

Accept multiple formats for dates, addresses, phone numbers, credit card numbers, and so on, per the Forgiving Format pattern. If a user does enter information that the form rejects, show an error message as soon as it becomes clear that the user made a mistake (you may need to wait until more form fields are filled out before deciding that for sure). On the form, politely indicate which input field is problematic, why, and how the user might fix it. See the patterns called Password Strength Meter and Same-Page Error Messages.

Beware a literal translation from the underlying programming model

Many forms are built to edit database records, or to edit objects in an object-oriented programming language. Given a data structure like these to fill out, it's really easy to design a form to do it. Each structure element gets (1) a label, and (2) a control (or a bundle of controls acting together). Put them in some rational order, lay them out top to bottom, and you're done, right?

Not entirely. This kind of implementation-driven form design does work, but it can give you a utilitarian and dull interface—or a difficult one. What if the structure elements don't match up with the input the user expects to give, for instance? And what if the structure is, say, 30 elements long? For some contexts, such as property sheets in a programming environment, it's appropriate to show everything the way it's implemented—that's part of the point. But for everything else, a more elegant and user-centered presentation is better.

So, here's the challenge: can you exploit dependencies among the structure elements, familiar graphic constructs (such as address labels), unstated assumptions, or knowledge of the user gained from previous interactions to make the form less onerous? Can you turn the problem into one handled by direct manipulation, such as dragging and dropping things around? Be creative!

Usability-test it

For some reason, when input forms are involved, it's particularly easy for designers and users to make radically different assumptions about terminology, possible answers, intrusiveness, and other context-of-use issues. This book has said it before, and will say it again: do some usability testing, even if you're reasonably sure your design is good. This will give you empirical evidence of what works and what doesn't for your particular situation.

Your choice of controls will affect the user's expectation of what is asked for, so choose wisely

A radio box suggests a one-of-many choice, while a one-line text field suggests a short, open-ended answer. Consciously or not, people will use the physical form of a control—its type, its size, and so forth—to figure out what's being asked for, and they'll set their expectations accordingly. If you use a text field to ask for a number, users may believe that any number is OK; if they enter "12" and you then surprise them with an error dialog box saying "The number you enter must be between 1 and 10," you've yanked the rug out from under them. A slider or spin box would have been better.

The following section gives you a table of possible controls for different input types. You or the engineers you work with will need to decide the semantics of each question. Is it binary? A date or time? One-of-many? Many-of-many? Open-ended but requiring validation? Look it up here, and then choose a control based on your particular design constraints.

Control Choice

The next sections describe controls and patterns for the kinds of information you might require from the user, such as numbers or choices from lists. It's not a complete set by any means; in fact, you can probably come up with plenty of others. But the types shown here are common, and the listed controls are among your best choices for clarity and usability.

Consider these factors when choosing among the possible controls for a given information type:

Available space
: Some controls take up lots of screen real estate; others are smaller, but may be harder to use than larger ones. Short forms on web pages might be able to spend that screen space on radio buttons or illustrated lists, whereas complex applications may need to pack as much content as possible into small spaces. Toolbars and table-style property sheets are especially constraining, since they generally allow for only one text line of height and often not much width, either.

User sophistication with respect to general computer usage
: Text fields should be familiar to almost all users of anything you'd design, but not everyone would be comfortable using a double-thumbed slider. For that matter, many occasional computer users don't know how to handle a multiple-selection listbox, either.

User sophistication with respect to domain knowledge
: A text field might be fine if your users know that, say, only the numbers 1–10 and 20–30 are valid. Beginners will stumble, but if they're a very small part of your user base (and if the context is readily learned), maybe that's OK—a tiny text field might be better than using a big set of interlinked controls.

Expectations from other applications
: For instance, Bold/Italic/Underline controls are known as iconic buttons; it would just be weird to see them as radio buttons instead.

Available technology
: As of this writing, HTML provides only a very small subset of the controls in common usage on the desktop: text fields, radio boxes and checkboxes, scrolled lists, and simple drop downs. Commercial and open source GUI toolkits provide richer sets of controls. Their offerings vary, but many of them are extensible via programming, allowing you to create custom controls for specific situations.

The following sections summarize the various control options for four common input scenarios: lists of items, text, numbers, and dates or times. Each choice is illustrated with a typical example, taken from the Windows 2000 look-and-feel. (Keep in mind that these examples are not necessarily the best possible rendering of these controls! You do have some freedom when you decide how to draw them, especially on the Web. See Chapter 9's introduction for further discussion.)

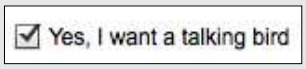
Lists of Items

A wide variety of familiar controls allow users to select items or options from lists. Your choice of control depends on the number of items or options to be selected (one or many) and the number of potentially selectable items (two, a handful, or many).

Here are controls for selecting one of two options (a binary choice).

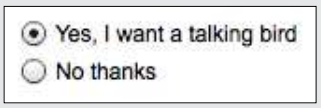
Checkbox

- Pros: simple; low space consumption
- Cons: can only express one choice, so its inverse remains implied and unstated; this can lead to confusion about what it means when it's off



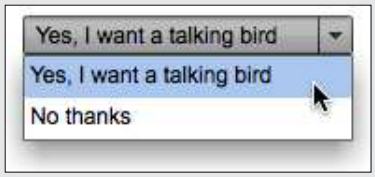
Two radio buttons

- Pros: both choices are stated and visible
- Cons: higher space consumption



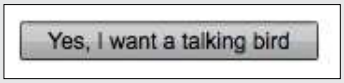
Two-choice drop-down list

- Pros: both choices are stated; low and predictable space consumption; easily expandable later to more than two choices
- Cons: only one choice is visible at a time; requires some dexterity



"Press-and-stick" toggle button

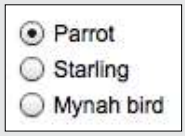
- Pros: same as for checkbox; when iconic, very low space consumption
- Cons: same as for checkbox; also, not as standard as a checkbox for text choices



The following controls are for selecting one of *N* items, where *N* is small.

N radio buttons

- Pros: all choices are always visible
- Cons: high space consumption

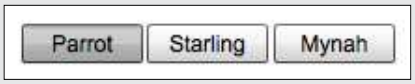


N-item drop-down list

- Pros: low space consumption
- Cons: only one choice is visible at a time, except when the menu is open; requires some dexterity

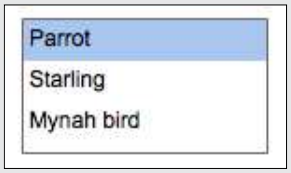
N-item set of mutually exclusive iconic toggle buttons

- Pros: low space consumption; all choices are visible
- Cons: icons might be cryptic, requiring tool tips for understanding; user might not know they're mutually exclusive



Single-selection list or table

- Pros: many choices are visible; can be kept as small as three items
- Cons: higher space consumption than drop-down list or spinner



Spinner

- Pros: low space consumption
- Cons: only one choice is ever visible at a time; requires a lot of dexterity; unfamiliar to naive computer users; drop-down list is usually a better choice

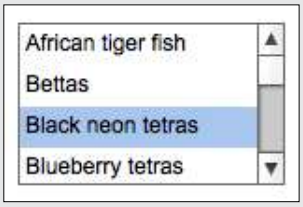
These controls are for selecting one of *N* items, where *N* is large.

N-item drop-down list, scrolled if necessary

- Pros: low space consumption
- Cons: only one choice is visible at a time, except when menu is open; requires a lot of dexterity to scroll through items on the drop-down menu

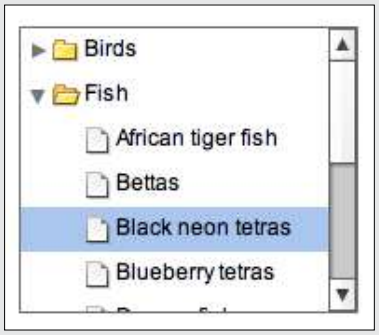
Single-selection list or table

- Pros: many choices are visible; can be kept small if needed
- Cons: higher space consumption than drop-down list



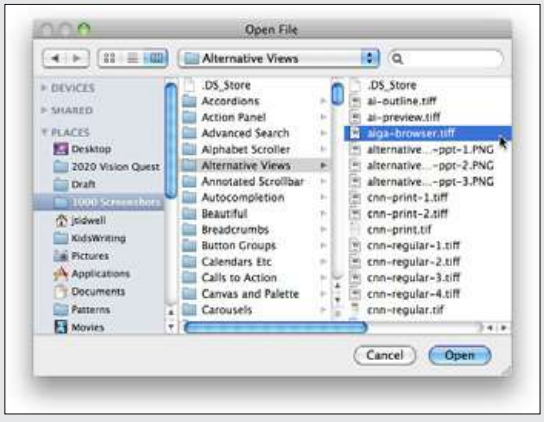
Single-selection tree or Cascading List, with items arranged into categories

- Pros: many choices are visible; organization helps findability in some cases
- Cons: may be unfamiliar to naive computer users; high space consumption; requires high dexterity



Custom browser, such as for files, colors, or fonts

- Pros: suited for browsing available choices
- Cons: may be unfamiliar to some users; difficult to design; usually a separate window, so it's less immediate than controls placed directly on the page



Here are controls for selecting many of *N* items, in any order.

Array of *N* checkboxes

- Pros: all choices are stated and visible
- Cons: high space consumption

Array of *N* toggle buttons

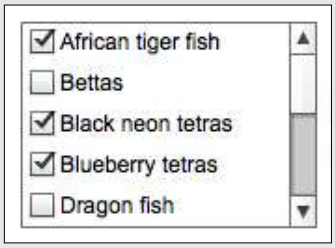
- Pros: low space consumption; all choices are visible
- Cons: icons might be cryptic, requiring tool tips for understanding; might look mutually exclusive

Multiple-selection list or table

- Pros: many choices are visible; can be kept small if needed
- Cons: not all choices are visible without scrolling; high (but bounded) space consumption; user might not realize it's multiple-selection

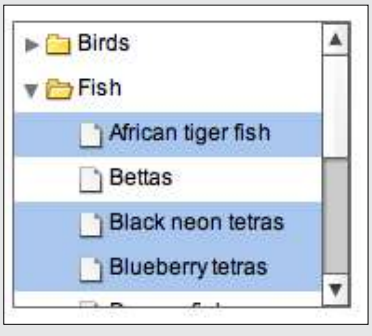
List with checkbox items

- Pros: many choices are visible; can be kept small if needed; affordance for selection is obvious
- Cons: not all choices are visible without scrolling; high (but bounded) space consumption



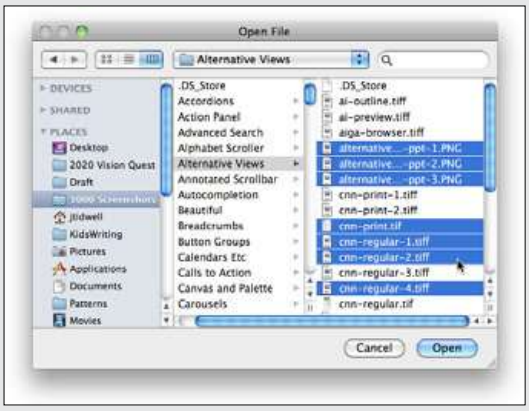
Multiple-selection tree or Cascading List, with items arranged into categories

- Pros: many choices are visible; organization helps findability in some cases
- Cons: may be unfamiliar to naive computer users; requires high dexterity; looks the same as single-selection tree



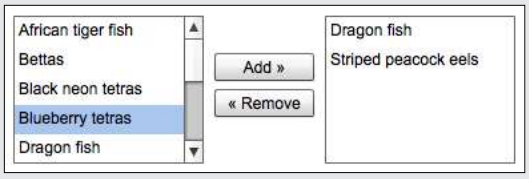
Custom browser, such as for files, colors, or fonts

- Pros: suited for browsing available choices
- Cons: may be unfamiliar to some users; difficult to design; usually a separate window, so it's less immediate than controls placed directly on the page



List Builder pattern

- Pros: selected set is easy to view; selection can be an ordered list if desired; easily handles a large source list
- Cons: very high space consumption due to two lists; does not easily handle a large set of selected objects



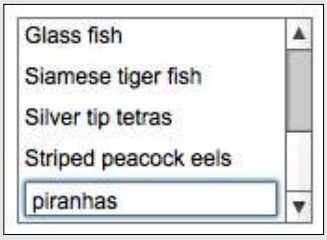
Use these controls for constructing an unordered list of user-entered items.

List or table with Add or New button

- Pros: "add" action is visible and obvious
- Cons: higher space consumption; visual clutter

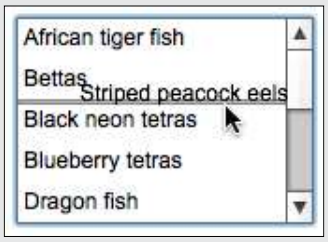
List or table with New-Item Row pattern

- Pros: lower space consumption; editing is done in place
- Cons: "add" action is not quite as obvious as a button



List or table that can receive dragged-and-dropped items

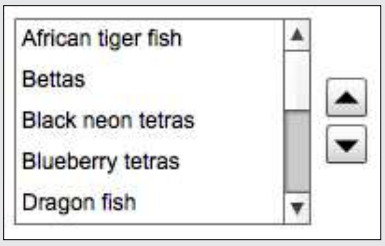
- Pros: visually elegant and space-saving; drag-and-drop is efficient and intuitive
- Cons: "add" action is not visible, so users may not know the list is a drop target



These controls are useful for constructing an ordered list of items.

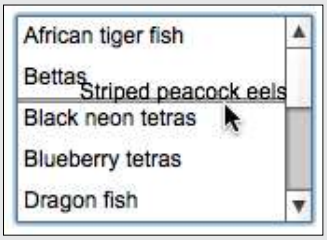
Unordered list with "up" and "down" affordances

- Pros: rearrangement actions are visible
- Cons: higher space consumption



Unordered list that offers internal drag-and-drop for reordering items

- Pros: visually elegant and space-saving; drag-and-drop is efficient and intuitive
- Cons: rearrangement actions are not visible, so users may not know they're available



Text

Collecting text input from a user is one of the most basic form tasks. The controls typically are chosen according to the number of lines to be entered, whether or not the lines are predetermined choices, and whether or not the text will include formatting.

The following control is for entering one line of text.

Single-line text field

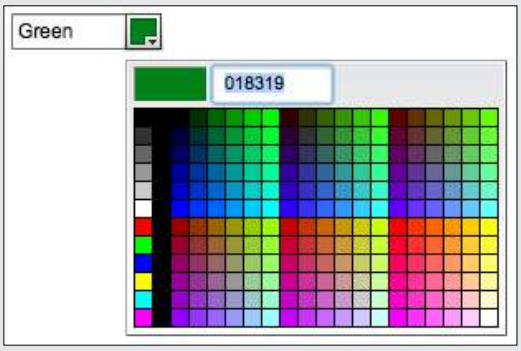
These controls are useful for entering either one line of text or a one-of-*N* choice.

Combo box

- Pros: quicker to use than a separate dialog box; familiar
- Cons: limited number of items can reasonably fit in drop down

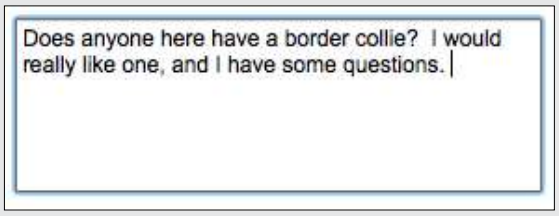
Text field with More button or Dropdown Chooser

- Pros: permits the launch of a specialized chooser dialog box, e.g., a file finder or drop down
- Cons: not as familiar as a combo box to some users; dialogs are not as immediate



This control is for entering multiple lines of unformatted text.

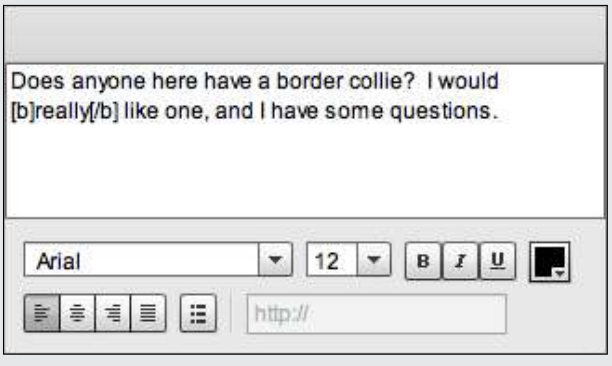
Multiline text area



These controls are for entering multiple lines of formatted text.

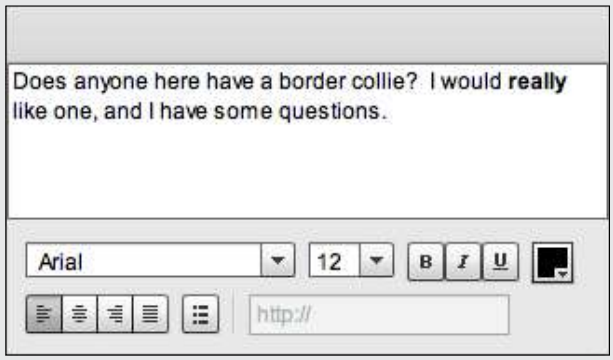
Text area with inline tags

- Pros: skilled users can avoid the toolbar by typing tags directly
- Cons: not truly WYSIWYG



Rich-text editor

- Pros: immediacy, since the edited text serves as a preview
- Cons: use of toolbar is required, so it cannot always be keyboard-only



Numbers

Because numbers often must follow more complex formatting rules, entering numbers on a form is slightly more complex than entering basic text. The choice of input options depends on the type of number you enter and its allowable range.

The following are controls for entering numbers of any type or format.

Text field using Forgiving Format

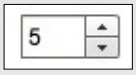
- Pros: visually elegant; permits wide variety of formats or data types
- Cons: expected format is not evident from the control's form, so it may cause temporary confusion; requires careful backend validation

Text field using Structured Format

- Pros: desired format evident from control's form
- Cons: possibly higher space consumption; more visual complexity; does not permit any deviation from the specified format, even if user needs to do so; may be more difficult for assistive technologies than a single field

Spin box (best for integers or discrete steps)

- Pros: user can arrive at a value via mouse clicks, without touching the keyboard; can also type directly if desired
- Cons: not familiar to all users; you may need to hold down the button long enough to reach the desired value; requires dexterity to use tiny buttons

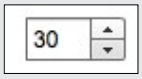


Use these controls for entering numbers from a bounded range.

Slider

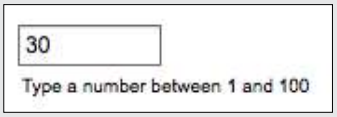
- Pros: obvious metaphor; position of value in range is shown visually; the user cannot enter a number outside the range
- Cons: high space consumption; unobvious keyboard access; tick labels can make it very crowded

Spinner



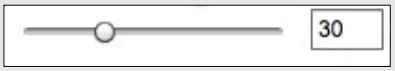
- Pros: values are constrained to be in range when buttons are used; low space consumption; supports both keyboard-only and mouse-only access
- Cons: not familiar to all users; requires dexterity to use tiny buttons; needs validation; cannot visually see value within range

Text field with after-the-fact error checking (can have Input Hints, Input Prompt, etc.)



- Pros: familiar to everyone; low space consumption; keyboard access
- Cons: requires validation; no constraints on what can be entered, so you have to communicate the range by some other means

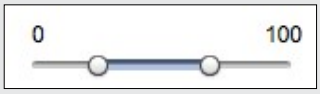
Slider with text field (can take the form of a Dropdown Chooser with a slider on the drop down)



- Pros: allows both visual and numeric forms of input
- Cons: complex; high space consumption when both elements are on the page; requires validation of text field when the user types the value

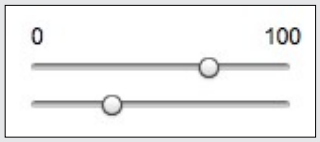
These controls are for entering a subrange from a larger range.

Double slider (can be used with two text fields)



- Pros: lower space consumption than two sliders
- Cons: unfamiliar to most users; no keyboard access unless you also use text fields

Two sliders (also can be used with text fields)

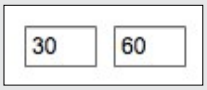


- Pros: less intimidating than a double slider
- Cons: very high space consumption; no keyboard access unless text fields are used, too

Two spinners (can be linked via Fill-in-the-Blanks)

- Pros: values are constrained to be in range when buttons are used; low space consumption; supports both keyboard-only and mouse-only access
- Cons: not familiar to all users; requires dexterity to use tiny buttons; needs validation; cannot visually see value within range

Two text fields with error checking (can use Input Hints, Input Prompt, or Fill-in-the-Blanks)

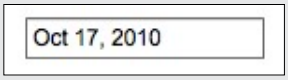


- Pros: familiar to everyone; much lower space consumption than sliders
- Cons: requires validation; no constraints on what can be entered, so you need to communicate the range by some other means

Dates or Times

Because of the potential formats and internationalization issues, dates and times can be a tricky item to accept from users. Input options for dates or times include the following.

Forgiving Format text field



- Pros: visually simple; permits wide variety of formats or data types; keyboard access
- Cons: expected format not evident from control's form, so it may cause brief confusion; requires careful backend validation

Structured Format text field

- Pros: desired format evident from control's form
- Cons: possibly higher space consumption; more visual clutter; does not permit deviation from specified format, even if user wants to do so; may be more difficult for screen readers than a single field

Calendar or clock control

- Pros: obvious metaphor; input is constrained to allowable values
- Cons: high space consumption; may not provide keyboard-only access

Dropdown Chooser with calendar or clock control

- Pros: combines the advantages of text field and calendar control; low space consumption
- Cons: complex interaction; requires dexterity to pick values from a drop down

The Patterns

As you might have guessed if you read through the control tables in the preceding section, most of these patterns describe controls—specifically, how you can combine controls with other controls and text in ways that make them easier to use. Some patterns define structural relationships between elements, such as Dropdown Chooser and Fill-in-the-Blanks. Others, such as Good Defaults and Autocompletion, discuss the values of controls and how those values change.

A large number of these patterns deal primarily with text fields: Forgiving Format, Structured Format, Fill-in-the-Blanks, Input Hints, Input Prompt, Password Strength Meter, and Autocompletion. That shouldn't be surprising. Text fields are as common as dirt, but they don't make it easy for users to figure out what should go in them. They're easiest to use when presented in a context that makes their usage clear. The patterns give you many ways to create that context.

1. Forgiving Format
2. Structured Format
3. Fill-in-the-Blanks
4. Input Hints
5. Input Prompt
6. Password Strength Meter
7. Autocompletion

The next two patterns deal with controls other than text fields. Dropdown Chooser describes a way to create a custom control, and List Builder, referenced in the control table shown earlier, describes a commonly reinvented combination of controls that lets users construct a list of items.

8. Dropdown Chooser
9. List Builder

You should design the remaining patterns into the whole form. They apply equally well to text fields, drop downs, radio buttons, lists, and other stateful controls, but you should use them consistently within a form (or within a dialog box, or even an entire application).

10. Good Defaults
11. Same-Page Error Messages

Patterns from other chapters apply to form design as well. From Chapter 4, Right/Left Alignment discusses one way to arrange labels alongside controls. Labels can also be placed above the form fields (at the cost of vertical space, but with plenty of horizontal room for long labels), or left-aligned along the left edge of the form. The choice can affect the speed of form completion.

Chapters 3 and 4 also give you some larger-scale design possibilities. A *gatekeeper form*—any form that stands between the user and his immediate goal, such as sign-up or purchase forms—should be in Center Stage, with very few distractions on the page. Alternatively, you might make it a Modal Panel, layered over the page.

If you have a long form that covers different topics, you might consider breaking it up into Titled Sections or even separate pages. (Tabs tend to work poorly as grouping mechanisms for forms.) If you break up a form into a sequence of pages, use the Wizard and Sequence Map patterns to show users where they are and where they're going.

Finally, forms should use a Prominent "Done" Button (Chapter 6) for the completion or submission action. If you have secondary actions, such as a form reset or a help link, make those less prominent.

Forgiving Format

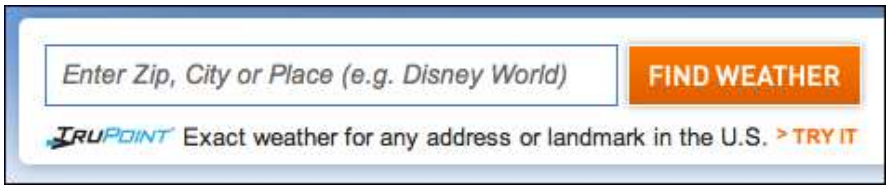


Figure 8-1. *Weather.com*

What

Permit users to enter text in a variety of formats and syntax, and make the application interpret it intelligently.

Use when

Your UI asks for data that users might type with an unpredictable mix of whitespace, hyphens, abbreviations, or capitalizations. More generally, the UI can accept input of various kinds from the user—different meanings, formats, or syntax. But you want to keep the interface visually simple.

Why

The user just wants to get something done, not think about "correct" formats and complex UIs. Computers are good at figuring out how to handle input of different types (up to a point, anyway). It's a perfect match: let the user type whatever he needs, and if it's reasonable, make the software do the right thing with it.

This can help simplify the UI tremendously, making it much easier to figure out. It may even remove the requirement for an Input Hint or Input Prompt, though they're often seen together, as in the example in Figure 8-1.

You might consider Structured Format as an alternative, but that pattern works best when the input format is entirely predictable (and usually numeric, like telephone numbers).

How

The catch (you knew there would be one): it turns a UI design problem into a programming problem. You have to think about what kinds of text a user is likely to type in. Maybe you ask for a date or time, and only the format varies—that's an easy case. Or maybe you ask for search terms, and the variation is what the software *does* with the data. That's harder. Can the software disambiguate one case from another? How?

Each application uses this pattern differently. Just make sure the software's response to various input formats matches what users expect it to do. Test, test, and test again with real users.

Examples

The *New York Times* uses Forgiving Format in several features that need information from users. Figure 8-2 shows examples from its real estate search and from its financial quotes feature.

Figure 8-2. *Two text fields in the New York Times website*

One place where this pattern should be used, but usually isn't, is when credit card numbers are requested from the user. As long as 16 digits are typed, why should the form care whether the user separates them by spaces, or by hyphens, or by nothing at all? It's not difficult to strip out separating characters. PayPal, for example, doesn't accept spaces in credit card numbers (see Figure 8-3).

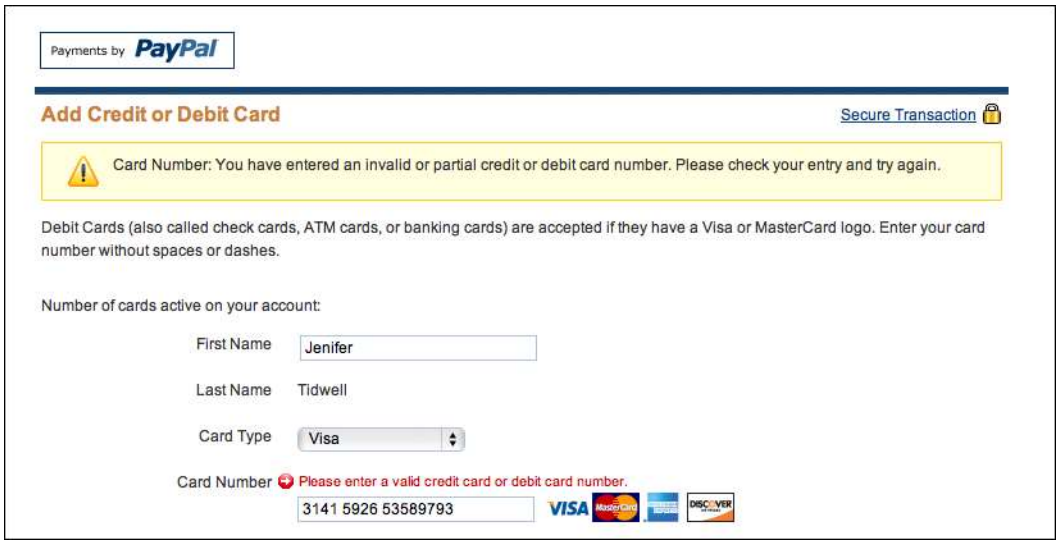


Figure 8-3. *PayPal*

Figure 8-4 comes from Outlook's tool for setting up a meeting. Look at the "Start time:" and "End time:" fields at the bottom of the screenshot—you don't need to give it a fully defined date, like what's in the text fields now. If today is April 24 and you want to set up a meeting for April 29, you can type any of the following terms:

- next Thu
- nxt thu
- thu
- 29/4/2004
- 4/29/2004
- 29/4
- 4/29
- five days
- 5 days

And so on—there are probably other accepted formats, too. The specified date then is "echoed back" to the user in the appropriate format for the user's language and location.

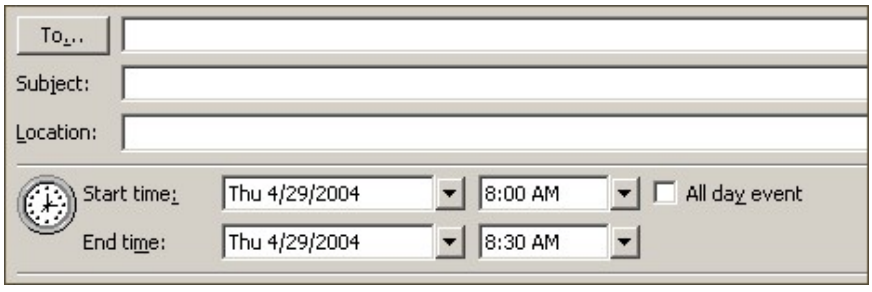


Figure 8-4. *Microsoft Outlook*

In other libraries

http://ui-patterns.com/patterns/ForgivingFormat

http://quince.infragistics.com/Patterns/Forgiving%20Format.aspx

Structured Format

Figure 8-5. *Photoshop installation screen*

What

Instead of using one text field, use a set of text fields that reflect the structure of the requested data.

Use when

Your interface requests a specific kind of text input from the user, formatted in a certain way. That format is familiar and well defined, and you don't expect any users to need to deviate from the format you expect. Examples include credit card information, local telephone numbers, and license strings or numbers.

It's generally a bad idea to use this pattern for any data in which the preferred format may vary from user to user. Consider especially what might happen if your interface is used in other countries. Names, addresses, postal codes, and telephone numbers all have different standard formats in different places. Consider using Forgiving Format in those cases.

Why

The structure of the text fields gives the user a clue about what kind of input is being requested. For example, she can mentally map the six text fields in Figure 8-5 to the six-chunk number written on her Photoshop CD case, and conclude that that's the license number she now needs to type in. Expectations are clear. She probably also realizes that she doesn't need to type in any spaces or hyphens to separate the six chunks.

This pattern usually gets implemented as a set of small text fields instead of one big one. That alone can reduce data entry errors. It's easier for someone to double-check several short strings (two to five characters or so) than one long one, especially when numbers are involved. Likewise, it's easier to transcribe or memorize a long number when it's broken up into chunks. That's how the human brain works.

Contrast this pattern to Forgiving Format, which takes the opposite tack: it allows you to type in data in any format, without providing structural evidence of what's being asked for. (You can use other clues instead, like Input Hints.) Structured Format is better for very predictable formats, Forgiving Format for open-ended input.

How

Design a set of text fields that reflect the format being asked for. Keep the text fields short, as clues to the length of the input.

Once the user has typed all the digits or characters in the first text field, confirm it for her by automatically moving the input focus to the next field. She can still go back and re-edit the first one, of course, but now she knows how many digits are required there.

You can also use Input Prompts to give the user yet more clues about what's expected. In fact, structured format fields for dates often do use Input Prompts, such as "dd/mm/yyyy".

Examples

At its simplest, Structured Format literally can take the shape of the data, complete with spaces, hyphens, and parentheses, as illustrated in the following table.

Telephone number	(504) 555-1212	(504) 555 - 1212
Credit card number	1021 1234 5678 0000	1021 1234 5678 0000
Date	12/25/2004	12 / 25 / 2004
ISBN number	0-1950-1919-9	0 - 1950 - 1919 - 9

For date input, LiveJournal uses Structured Format in combination with a drop down to choose a month (see Figure 8-6). It defaults to the current day and time.

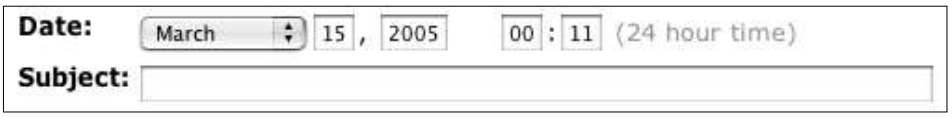


Figure 8-6. *LiveJournal*

In other libraries

http://ui-patterns.com/patterns/StructuredFormat

http://quince.infragistics.com/Patterns/Structured%20Format.aspx

Fill-in-the-Blanks

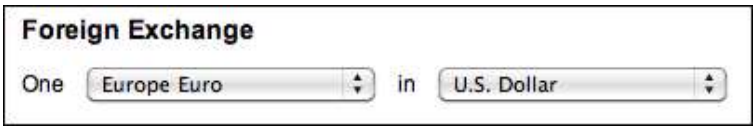


Figure 8-7. *The New York Times*

What

Arrange one or more fields in the form of a prose sentence or phrase, with the fields as "blanks" to be filled in by the user.

Use when

You need to ask the user for input, usually one-line text, a number, or a choice from a drop-down list. You tried to write it out as a set of label/control pairs, but the labels' typical declarative style (such as "Name:" and "Address:") isn't clear enough for users to understand what's going on. You can, however, verbally describe the action to be taken once everything's filled out, in an active-voice sentence or phrase.

Why

Fill-in-the-Blanks helps to make the interface self-explanatory. After all, we all know how to finish a sentence. (A verb phrase or noun phrase will do the trick, too.) Seeing the input, or "blanks," in the context of a verbal description helps the user understand what's going on and what's being asked of him.

How

Write the sentence or phrase using all your word-crafting skills. Use controls in place of words.

If you're going to embed the controls in the middle of the phrase instead of at the end, this pattern works best with text fields, drop-down lists, and combo boxes—in other words, controls with the same form factor (width and height) as words in the sentence. Also, make sure the baseline of the sentence text lines up with the text baselines in the controls, or it'll look sloppy. Size the controls so that they are just long enough to contain the user's choices, and maintain proper word spacing between them and the surrounding words.

This is particularly useful for defining conditions, as one might do when searching for items or filtering them out of a display. The Excel and eBay examples in Figures 8-8 and 8-9 illustrate the point. Robert Reimann and Alan Cooper describe this pattern as an ideal way to handle queries; their term for it is *natural language output*.*

There's a big "gotcha" in this pattern, however: it becomes very hard to properly *localize* the interface (convert it to a different language), since comprehension now depends upon word order in a natural language. For some international products or websites, that's a nonstarter. You may have to rearrange the UI to make it work in a different language; at the very least, work with a competent translator to make sure the UI can be localized.

Examples

The Excel cell-formatting dialog box shown in Figure 8-8 lets you choose the phrases in this "sentence" from drop-down boxes. As the phrases change, the subsequent text fields—showing 0.7 and 0.9 in this example—might be replaced by other controls, such as a single text field for "greater than."

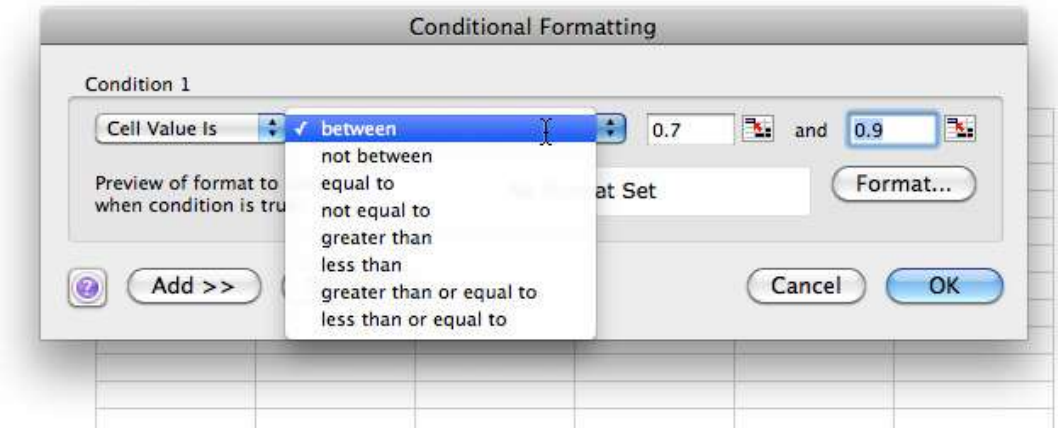


Figure 8-8. *Excel*

When users search for items on eBay, they can use a long form to filter search results according to various criteria. The form shown in Figure 8-9 has several examples of Fill-in-the-Blanks.

* See their book *About Face 2.0: The Essentials of Interaction Design* (Wiley), page 205.

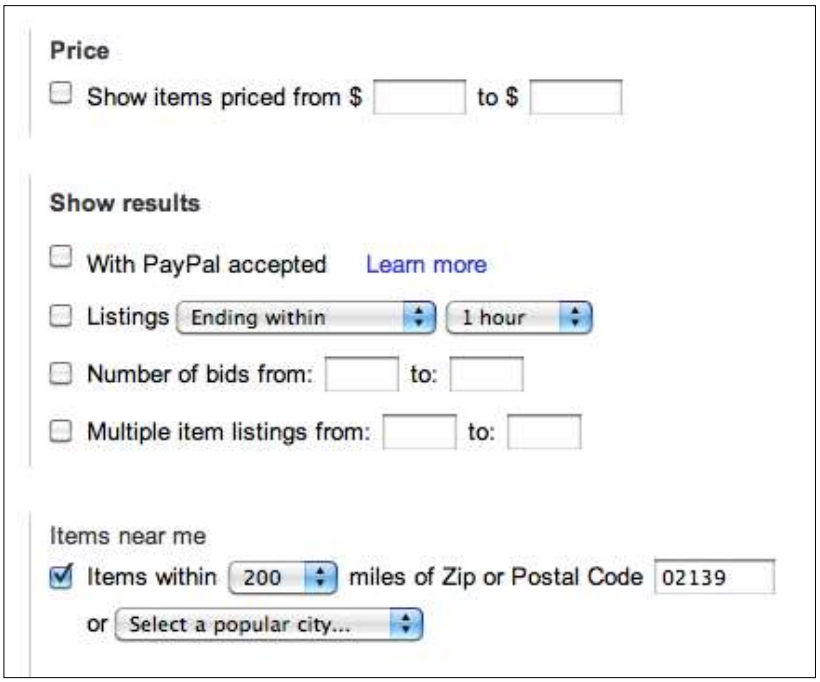


Figure 8-9. *eBay search filter form*

In other libraries

http://ui-patterns.com/patterns/FillInTheBlanks

Input Hints

Figure 8-10. *Twitter registration page*

What

Beside or below an empty text field, place a phrase or example that explains what is required.

Use when

The interface presents a text field, but the kind of input it requires isn't obvious to all users. You don't want to put more than a few words into the text field's label.

Why

A text field that explains what goes into it frees users from having to guess. The hint provides context that the label itself may not provide. If you visually separate the hint from the main label, users who know what to do can more or less ignore the hint, and stay focused on the label and control.

How

Write a short example or explanatory sentence, and put it below or beside the text field. The hint may be visible all the time, or it may appear when the text field receives input focus.

Keep the text in the hint small and inconspicuous, though readable; consider using a font two points smaller than the label font. (A one-point difference will look more like a mistake than an intended font-size change.) Also, keep the hint short. Beyond a sentence or two, many users' eyes will glaze over, and they'll ignore the text altogether.

This pattern is often used in conjunction with Forgiving Format, as illustrated by the Word example in Figure 8-11, or Structured Format. Alternative solutions include Input Prompt (in which a short hint goes into the control itself), tool tips that show a description on hover, and Good Defaults (which put an actual valid value into the control). The advantage of Input Hints is that it leaves the control blank—the user is forced to consider the question and give an answer, which is sometimes better than letting the user not think about it at all.

Examples

The printing dialog boxes used by several Microsoft Office applications supply an Input Hint below a Forgiving Format text field—it takes page numbers, page ranges, or both (Figure 8-11). The hint is very useful to anyone who's never had to use the "Pages" option, but users who already understand it don't need to focus on the written text; they can just go straight for the input field.

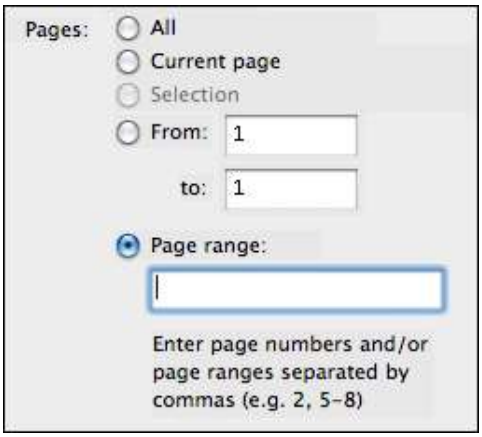


Figure 8-11. *Microsoft Word print dialog*

Longer descriptions can be used in Input Hints when necessary. The examples from Gmail's registration page, shown in Figure 8-12, are about as long as you'd want to put next to a text field—a user can click the links for further information. But most users will never follow a link when they're filling out a form, especially if they're trying to get through it quickly and don't have major privacy concerns, so don't depend on linked pages to convey critical information.

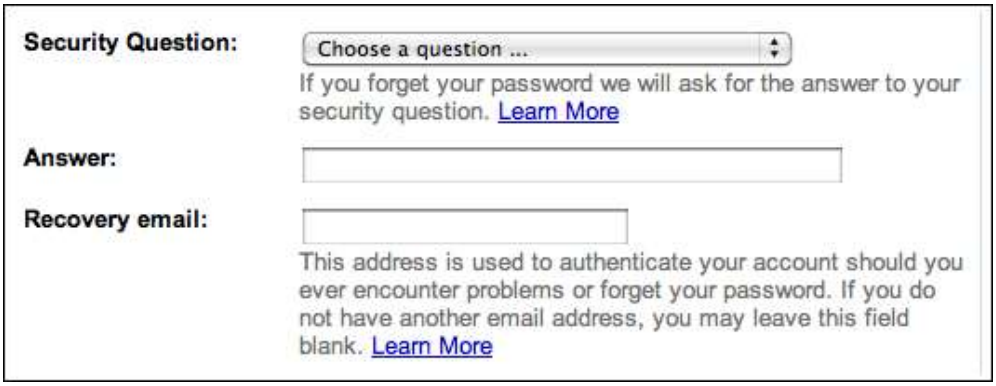


Figure 8-12. *Gmail registration page*

Blogger places Input Hints on the far right of the form, with horizontal rules that align the controls with their hints (see Figure 8-13). This is a graceful way to structure a page full of Input Hints.

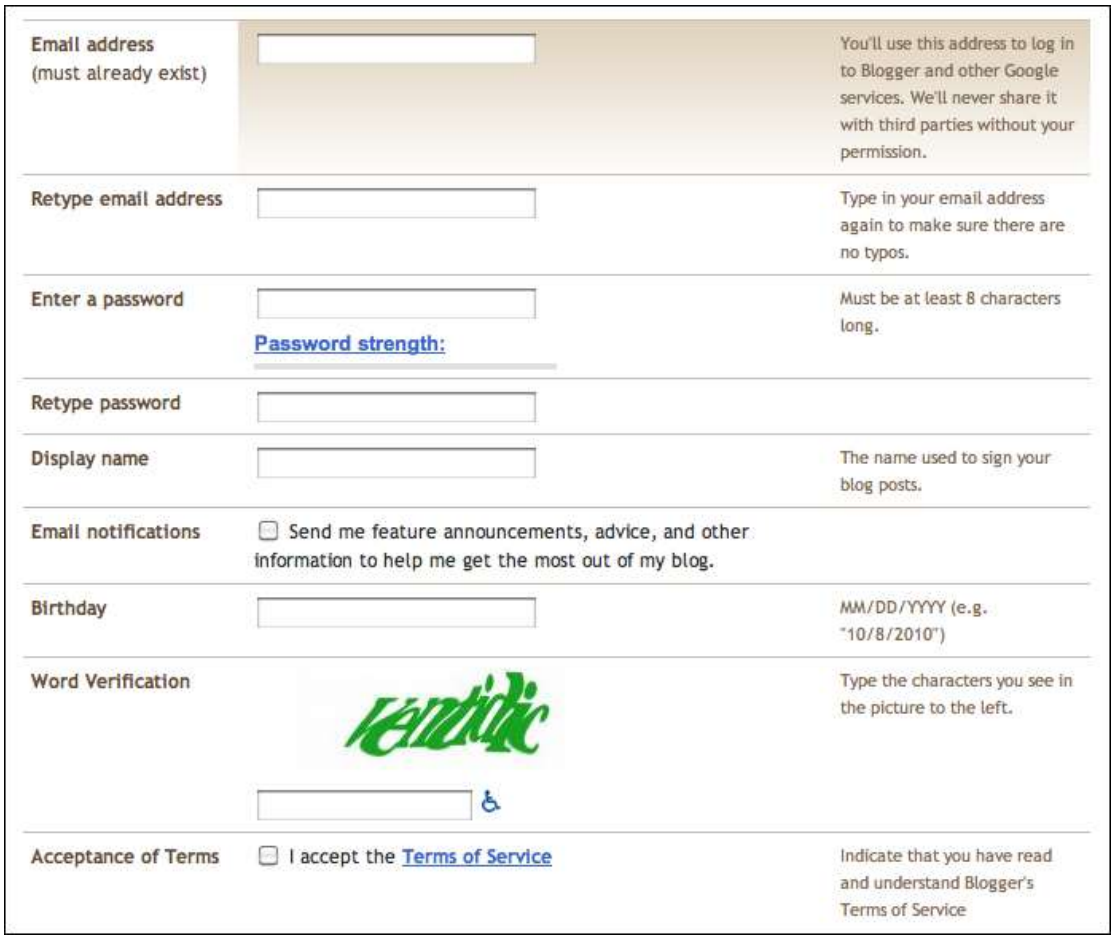


Figure 8-13. *Blogger registration page*

Some forms show an Input Hint when the user puts input focus into the text field itself (see Figure 8-14). This is nice because the hidden hints don't clutter the interface or add visual noise; however, the user doesn't see them at all until he clicks on (or tabs into) the text field. If you use these, note that you must leave space for them in the interface, just as you would for Hover Tools (Chapter 6). Twitter, shown first in this example, uses both kinds.

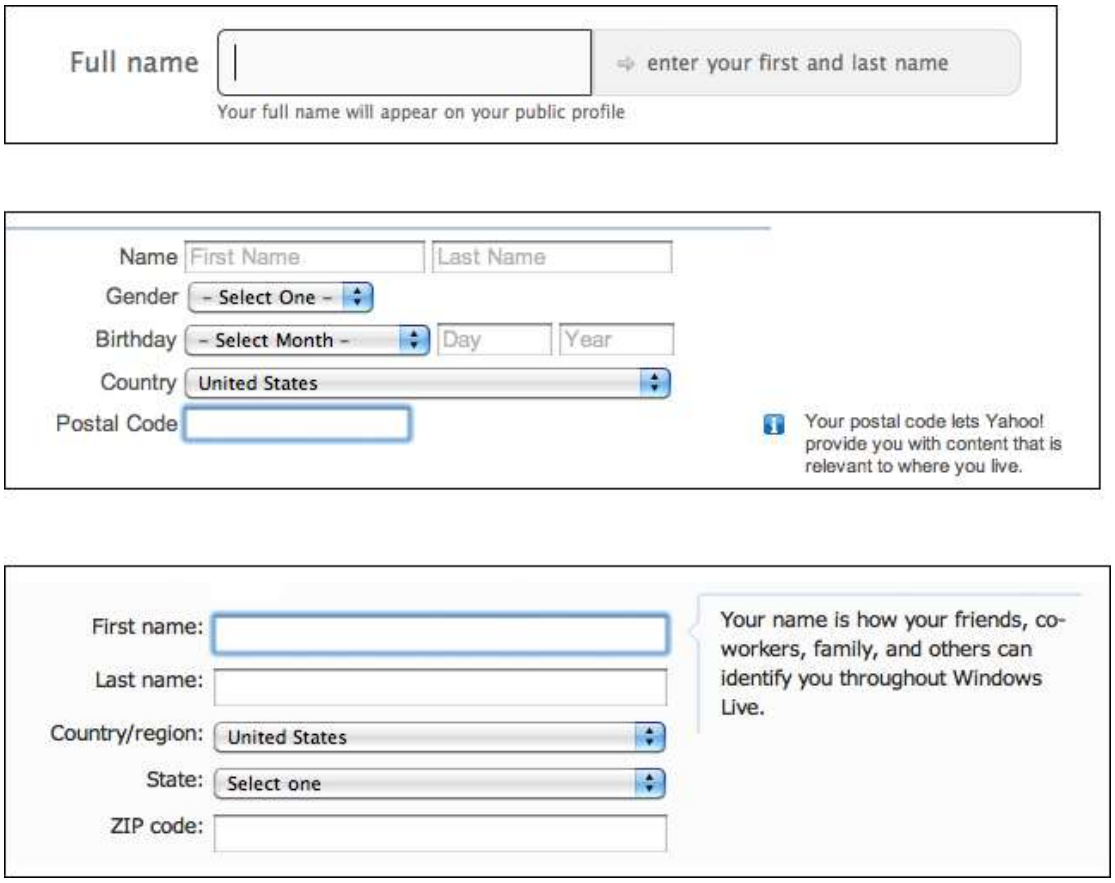


Figure 8-14. *Twitter, Yahoo!, and Hotmail registration pages, all with dynamic Input Hints*

In other libraries

http://quince.infragistics.com/Patterns/Input%20Hints.aspx

Input Prompt

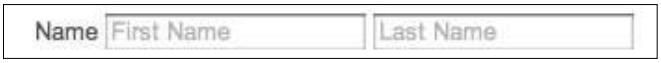


Figure 8-15. *Yahoo! registration page*

What

Prefill a text field or drop down with a prompt that tells the user what to do or type.

Use when

The UI presents a text field, drop down, or combo box for input. Normally you would use a good default value, but you can't in this case—perhaps there is no reasonable default, as in the Yahoo! form in Figure 8-15.

Why

It helps make the UI self-explanatory. Like Input Hints, an Input Prompt is a sneaky way of supplying help information for controls whose purpose or format may not be immediately clear.

With an Input Hint, someone quickly scanning the UI can easily ignore the hint (or miss it entirely). Sometimes this is your desired outcome. But an Input Prompt sits right where the user will type, so it can't be ignored. The advantage here is that the user doesn't have to guess whether she has to deal with this control or not—the control itself tells her she does. (Remember that users don't fill out forms for fun—they'll do as little as needed to finish up and get out of there.) A question or an imperative "Fill me in!" is likely to be noticed.

An interesting side effect of this pattern is that users may not even bother to read the label that prefixes the text field. Look again at Figure 8-15. The label "Name" is now completely superfluous, in terms of the form's meaning. Because the user's eye will be drawn first to the text fields, the "First Name" and "Last Name" prompts probably will be read before the "Name" label anyway! That being said, don't remove the labels—that prompt is gone once the user types over it, and on subsequent readings of this form, she may not remember what the control asks for.

If you're very careful to implement this pattern correctly, you may be able to do away with the label altogether. The prompt must be put back when the user erases the value, and the requested information must be very familiar to the user (such as name or email).

How

Choose an appropriate prompt string, perhaps beginning with one of these words:

- For a drop-down list, use *Select*, *Choose*, or *Pick*.
- For a text field, use *Type* or *Enter*.

End it with a noun describing what the input is, such as "Choose a state," "Type your message here," or "Enter the patient's name." Put this phrase into the control where the value would normally be. (The prompt itself shouldn't be a selectable value in a drop down; if the user selects it, it's not clear what the software should do with it.)

Since the point of the exercise was to tell the users what they were required to do before proceeding, don't let the operation proceed until they've done it! As long as the prompt is still sitting untouched in the control, disable the button (or other device) that lets the user finish this part of the operation. That way, you won't have to throw an error message at the user.

For text fields, put the prompt back into the field as soon as the user erases the typed response.

Use Good Defaults instead of an Input Prompt when you can make a very accurate guess about what value the user will put in. The user's email address may already have been typed somewhere else, for instance, and the originating country can often be detected by websites.

Examples

Figures 8-16 and 8-17 are two examples of forms that depend on the Input Prompt, in the absence of actual labels. Both are asking for very simple, well-understood answers that users should be able to type or select without thinking very hard about them. Both put the Input Prompt back into the field if there is no user-typed value, as shown by the second screenshot in each example. (Apple then turns the field yellow to reinforce that the value is required to complete the form; this is a gentle variant of Same-Page Error Messages.)

The Culinary Culture example demonstrates the striking visual look that can be achieved with a skillful combination of typography, icon design, and Input Prompt.

Figure 8-16. *Apple's purchase form*

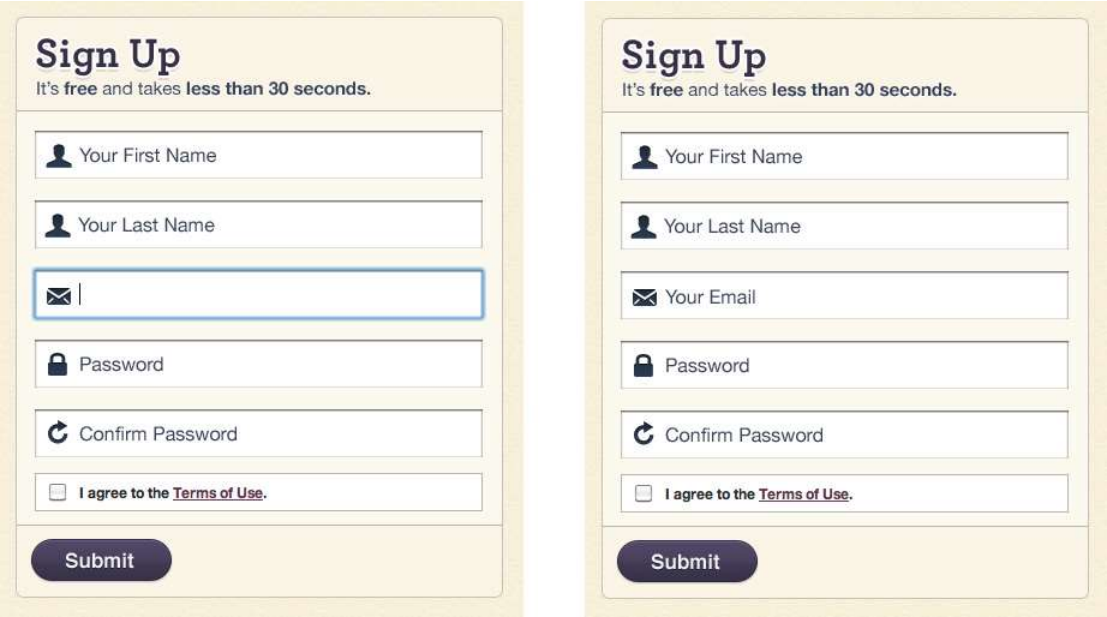


Figure 8-17. *CulinaryCulture.com*

In other libraries

http://quince.infragistics.com/Patterns/Input%20Prompt.aspx

http://ui-patterns.com/patterns/InputPrompt

Password Strength Meter

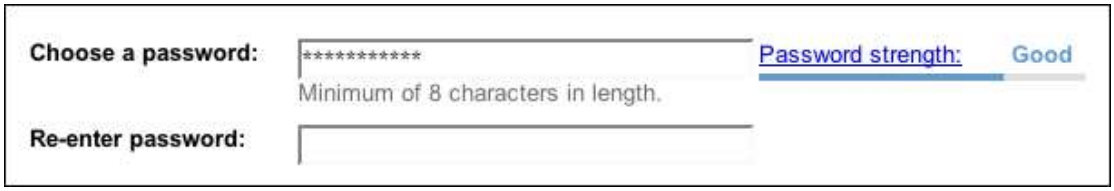


Figure 8-18. *Gmail registration page*

What

Give the user immediate feedback on the validity and strength of a new password while it is being typed.

Use when

The UI asks the user to choose a new password. This is quite common for site registrations. Your site or system cares about having strong passwords, and you want to actively help users choose good ones.

Why

Strong passwords protect both the individual user and the entire site, especially when the site handles sensitive information and/or social interactions. Weak passwords ought to be disallowed because they permit break-ins.

A Password Strength Meter gives immediate feedback to the user about his new password—is it strong enough or not? Does he need to make up a new one, and if so, with what characteristics (numbers, capital letters, etc.)? If your system is going to reject weak passwords, it's usually best to do it instantly, not after the user has submitted the registration form.

How

While the user types his new password, or after keyboard focus leaves the text field, show an estimate of the password strength beside the text field. At minimum, display a text and/or graphic label indicating a weak, medium, or strong password, and special wording to describe a too-short or invalid password. Colors help: red for unacceptable, green or blue for good, and some other color (often yellow) in between.

If you can, show additional text with specific advice on how to make a weak password better—a minimum length of eight characters (for instance), or the inclusion of numbers or capital letters. A user might get frustrated if he repeatedly fails to produce a valid password, so help him be successful.

Also, the form containing the password field should use Input Hints or other text to explain this beforehand. A short reminder of good password heuristics can be useful to users who need reminders, and if your system will actually reject weak passwords, you should warn the user about it before he finishes the form! Many systems require a minimum number of characters for a valid password, such as six or eight.

(Remember, never actually show a password, and don't make suggestions of alternative passwords. General hints are all you can really give.)

An explanation of password security is beyond the scope of a UI pattern. There are excellent online and print references for this topic, however, should you need to understand it more deeply.

Examples

Blogger's Password Strength Meter, shown in Figure 8-19, displays five states, one of which ("Too short") tells the user specifically how to fix the password—eight characters are required. The blue link puts up a window describing how to create a strong password, and there is an Input Hint (not shown) on the right side that tells the user about the eight-character minimum.

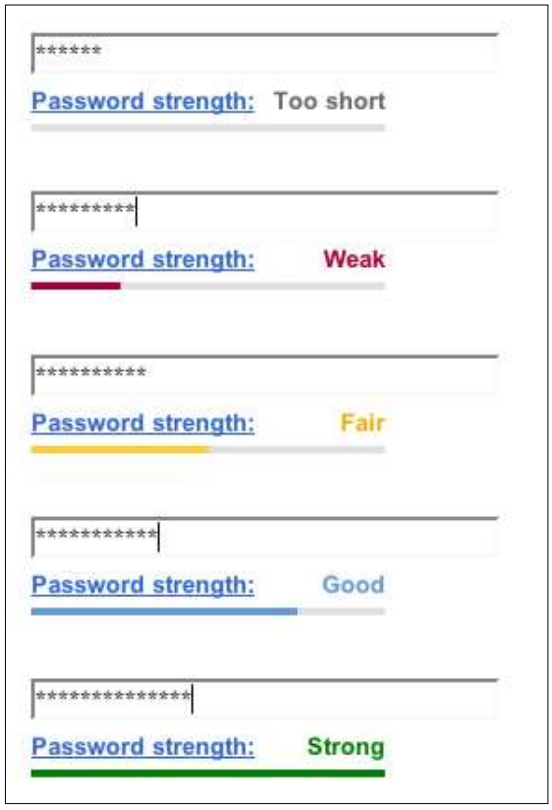


Figure 8-19. *Blogger's five states*

MSN shows only three states (see Figure 8-20). It also uses an Input Hint to describe the minimum—"Six-character minimum with no spaces"—and offers a link to a more detailed explanation. This meter is visually more heavyweight than Blogger's.

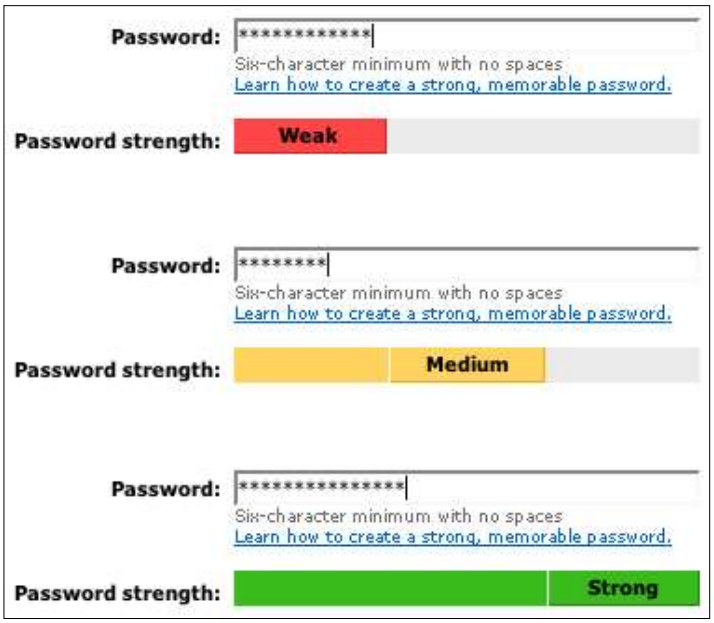


Figure 8-20. *MSN's three states*

Yahoo! offers specific, detailed password advice in two different Input Hints that appear when the password field received input focus (see Figure 8-21).

Figure 8-21. *Yahoo!*

In other libraries

http://ui-patterns.com/patterns/PasswordStrengthMeter

Code to do password checking is available for JavaScript and other languages. Look online not just for the term *password strength meter* but also *password meter*, *password checker*, and other variations.

Autocompletion

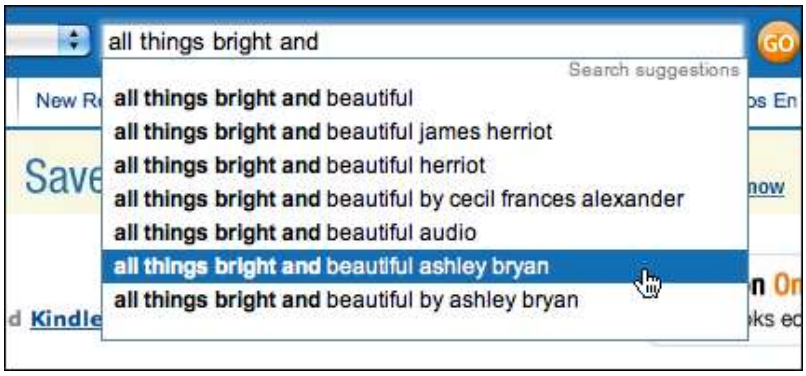


Figure 8-22. *Amazon*

What

As the user types into a text field, anticipate the possible answers, show a selectable list of them, and automatically complete the entry when appropriate.

Use when

The user types something predictable, such as a URL, the user's own name or address, today's date, or a filename. You can make a reasonable guess as to what she's attempting to type—perhaps there's a saved history of things this user has previously typed, for instance, or perhaps she is picking from a set of preexisting values, such as a list of filenames in a directory.

Search boxes, browser URL fields, email fields, common web forms (such as site registration or purchase), text editors, and command lines all seem to be much easier to use when supported by Autocompletion.

Why

Autocompletion saves time, energy, cognitive burden, and wrist strain for the user. It turns a laborious typing effort into a simple pick list (or less, if a single completion can be reliably supplied). You can thus save your users countless seconds of work, and contribute to the good health of thousands of wrists.

When the typed entries are long and hard to type (or remember), like URLs or email addresses, Autocompletion is quite valuable. It reduces a user's memory burden by supplying "knowledge in the world" in the form of a drop-down list. An additional benefit can be error prevention: the longer or stranger the string that must be typed, the greater the odds of the user making a typographical error. Autocompleted entries have no such problems.

For mobile devices, it's even more valuable. Typing text on a tiny device is no fun; if a user needs to enter a long string of letters, appropriate Autocompletion can save her a great deal of time and frustration. Again, email addresses and URLs are excellent candidates, to support mobile email and web usage.

Autocompletion is also common in text editors and command-line UIs. As users type commands or phrases, the application or shell might offer suggestions for completion. Code editors and OS shells are well suited for this, because the language used is limited and predictable (as opposed to a human language, such as English); it's therefore easier to guess what the user tries to type.

Finally, lists of possible autocompletions can serve as a map or guide to a large world of content. Search engines and site-wide search boxes do this well—when the user types the beginning of a phrase, an Autocompletion drop down shows likely completions that other people have typed (or that refer to available content). Thus, small corrections and gentle guidance are provided to a curious or uncertain user, and they offer a way to navigate a small corner of the public mental landscape.

How

With each additional character that the user types, the software quietly forms a list of the possible completions to that partially entered string. If the user enters one of a limited number of possible valid values, use that set of valid values. If the possible values are wide open, one of these might supply completions:

- Previous entries typed by this user, stored in a preferences or history mechanism
- Common phrases that many users have used in the past, supplied as a built-in "dictionary" for the application
- Possible matches drawn from the content being searched or perused, as for a site-wide search box
- Other artifacts appropriate to the context, such as company-wide contact lists for internal email

From here, you can approach the interaction design of Autocompletion in two ways. One is to show the user a list of possible completions on demand—for example, by pressing the Tab key—and let him choose one explicitly by picking from that list. Many code editors do this (see Figure 8-26 in the Examples section). It's probably better used when the user would recognize what he wants when he sees it, but may not remember how to type it without help. "Knowledge in the world is better than knowledge in the head."

The other way is to wait until there's only one reasonable completion, and then put it in front of the user, unprompted. Word does this with a tool tip; many forms do it by filling in the remainder of the entry but with selection turned on, so another keystroke would wipe out the autocompleted part. Either way, the user gets a choice about whether to retain the Autocompletion or not—and the default is to not keep it.

You can use both approaches together, as in Figure 8-26.

Make sure that Autocompletion doesn't irritate users. If you guess wrong, the user won't like it—he then has to erase the Autocompletion and retype what he meant in the first place, avoiding having Autocompletion pick the wrong completion yet again. These interaction details can help prevent irritation:

- Always give the user a choice to take the completion or not take it; default to "no."
- Don't interfere with ordinary typing. If the user intends to type a certain string and just keeps typing in spite of the attempts at Autocompletion, make sure the result is what the user intended to type.
- If the user keeps rejecting a certain Autocompletion in one place, don't keep offering it. Let it go at some point.
- Guess correctly.

Here's one possible way to implement Autocompletion cheaply. You can turn a text field into a combo box (which is a combination of a typable text field and a drop down). Each time the user enters a unique value into the text field, make a new drop-down item for it. Now, if your GUI toolkit allows type-ahead in combo boxes (as many do), the drop-down items are automatically used to complete whatever the user types. Refer back to Figure 8-22 at the top of the pattern for a typical example; most web browsers now keep the most recently visited sites in a combo box where the user types URLs.

Examples

Many email clients, of course, use Autocompletion to help users fill in To: and CC: fields. They generally draw on an address book, contacts list, or list of addresses you've exchanged email with. The example from Mac Mail, shown in Figure 8-23, shows a single completion suggested upon typing the letter *c*; the completed text is automatically highlighted, so a single keystroke can get rid of it. You can thus type straight "through" the completion if it's wrong.

Figure 8-23. *Mac Mail*

Drop-down lists of Autocompletion possibilities can take many forms. Figure 8-24 shows several examples of drop-down list formatting.

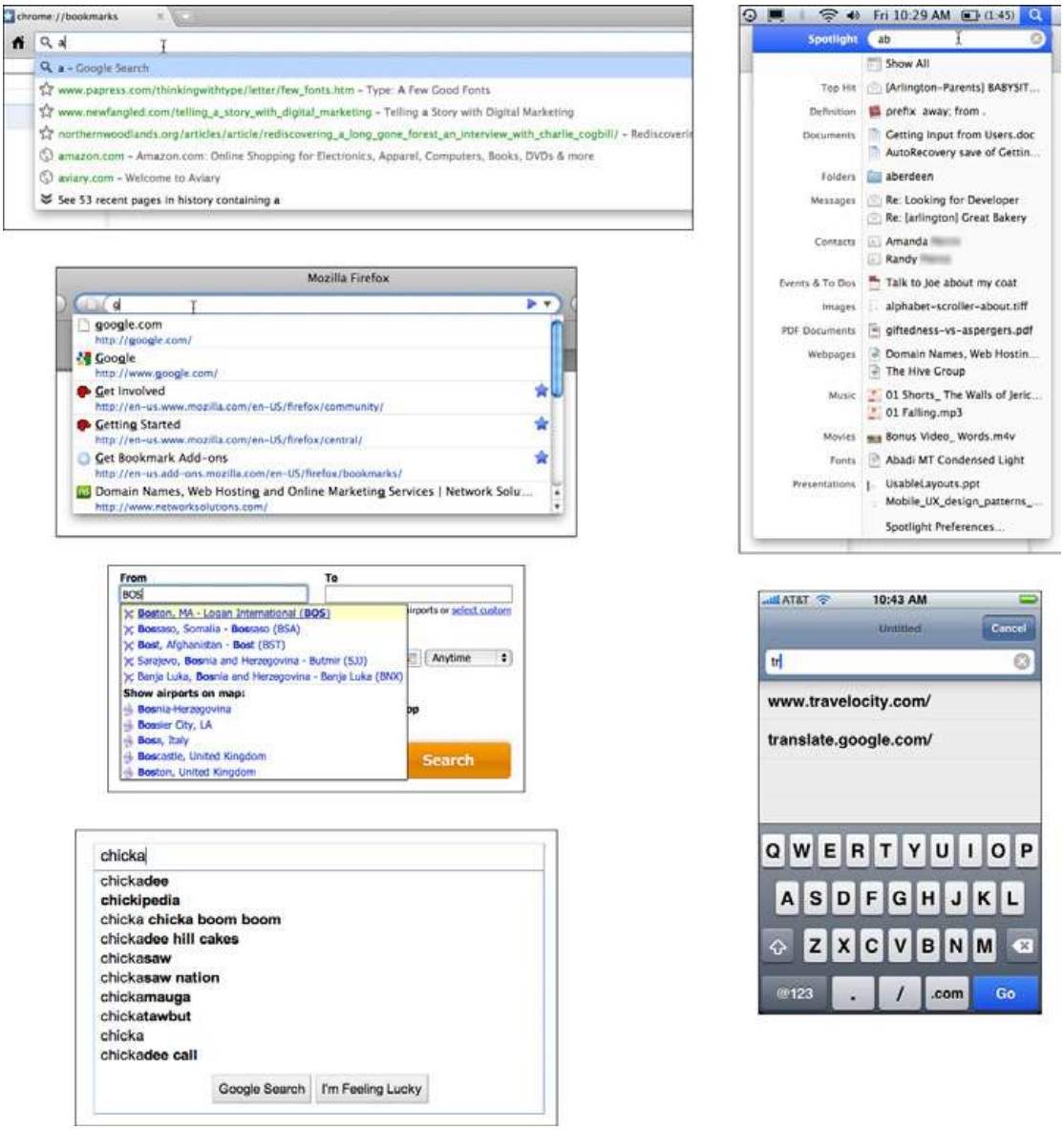


Figure 8-24. *Counterclockwise from top left: Chrome, Firefox, Kayak, Google, Safari for iPhone, and Mac OS Spotlight*

Dopplr, shown in Figure 8-25, doesn't show the whole long list of completions. Instead, it simply tells the user that there are 40 possible completions (for instance), and puts them behind a link.

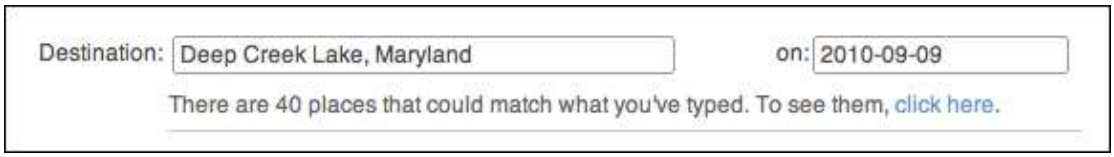


Figure 8-25. *Dopplr*

Finally, code editors such as Visual Studio invest in very complex Autocompletion mechanisms (see Figure 8-26). Visual Studio's IntelliSense completes the built-in keywords of a programming language, of course, but it also draws on the functions, classes, and variable names defined by the user. It even can show the arguments to functions that you invoke (in the righthand screenshot). Furthermore, both "select from a list" and "take the one completion that matches" approaches are supported, and you can call up Autocompletion on demand by pressing Ctrl-space bar.

Autocompletion in Visual Studio thus serves as a typing aid, a memory aid, and a browser of context-appropriate functions and classes. It's very useful.

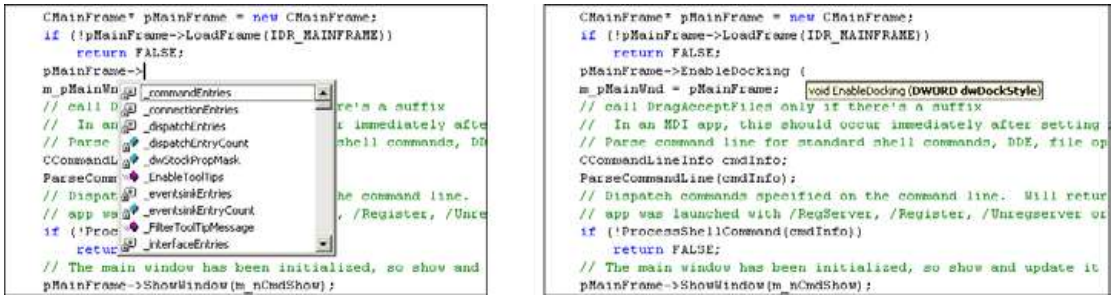


Figure 8-26. *Visual Studio*

In other libraries

http://developer.yahoo.com/ypatterns/selection/autocomplete.html

http://ui-patterns.com/patterns/Autocomplete

http://patternry.com/p=autocomplete/

http://www.welie.com/patterns/showPattern.php?patternID=autocomplete

(Note that most other libraries call this pattern "Autocomplete.")

Dropdown Chooser

Figure 8-27. *Microsoft Word*

What

Extend the concept of a menu by using a drop-down or pop-up panel to contain a more complex value-selection UI.

Use when

The user needs to supply input that is a choice from a set (such as in the color example in Figure 8-27), a date or time, a number, or anything other than free text typed at a keyboard. You want to provide a UI that supports that choice—a nice visual rendering of the choices, for instance, or interactive tools—but you don't want to use space on the main page for that; a tiny space showing the current value is all you want.

Why

Most users are very familiar with the drop-down list control (called a "combo box" when used with a free-typing text field). Many applications successfully extend this concept to drop downs that aren't simple lists, such as trees, 2D grids, and arbitrary layouts. Users seem to understand them with no problem, as long as the controls have down-arrow buttons to indicate that they open when clicked.

Dropdown Choosers encapsulate complex UIs in a small space, so they are a fine solution for many situations. Toolbars, forms, dialog boxes, and web pages of all sorts use them now. The page the user sees remains simple and elegant, and the chooser UI only shows itself when the user requests it—an appropriate way to hide complexity until it is needed.

How

For the Dropdown Chooser control's "closed" state, show the current value of the control in either a button or a text field. To its right, put a down arrow. This may be in its own button or not, as you see fit; experiment and see what looks good and makes sense to your users. A click on the arrow (or the whole control) brings up the chooser panel, and a second click closes it again.

Design a chooser panel for the choice the user needs to make. Make it relatively small and compact; its visual organization should be a familiar format, such as a list, a table, an outline-type tree, or a specialized format like a calendar or calculator (see the examples in the next section). See Chapter 5 for a discussion of list presentation.

Scrolling the panel is OK if the user understands that it's a choice from a large set, such as a file from a filesystem, but keep in mind that scrolling one of these pop-up panels is not easy for people without perfect dexterity!

Links or buttons on the panel can in turn bring up secondary UIs—for example, color-chooser dialog boxes, file-finder dialog boxes, or help pages—that help the user choose a value. These devices usually are modal dialog boxes. In fact, if you intend to use one of these modal dialogs as the primary way the user picks a value (say, by launching it from a button), you could use a Dropdown Chooser instead of going straight to the modal dialog. The pop-up panel could contain the most common or recently chosen items. By making frequently chosen items so easy to pick, you reduce the total time (or number of clicks) it takes for an average user to pick values.

Examples

Photoshop's compact, interaction-rich toolbars use Dropdown Choosers heavily. Two examples, Brush and Opacity, are shown in Figure 8-28. The Brush chooser is a selectable list with a twist—it has extra controls such as a slider, a text field, and a pull-right button (the circular one) for yet more choices. The Opacity chooser is a simple slider, and the text field above it echoes its value.

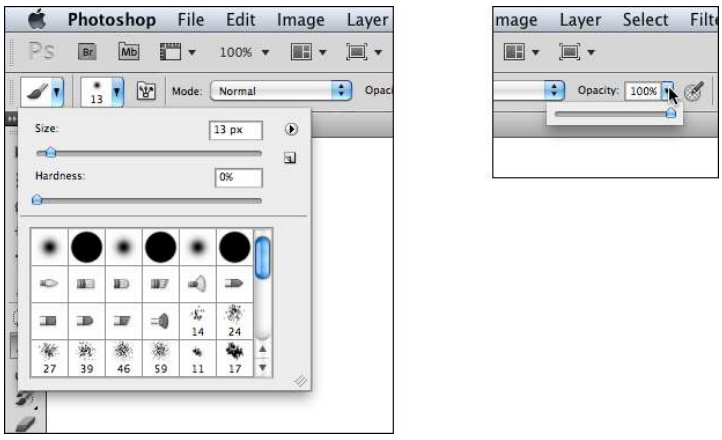


Figure 8-28. *Photoshop drop downs*

The Thumbnail Grid pattern (Chapter 5) is often used in Dropdown Choosers in place of a text-based menu. The examples from PowerPoint (Figure 8-29) and iWeb (Figure 8-30) demonstrate two styles of Thumbnail Grid.

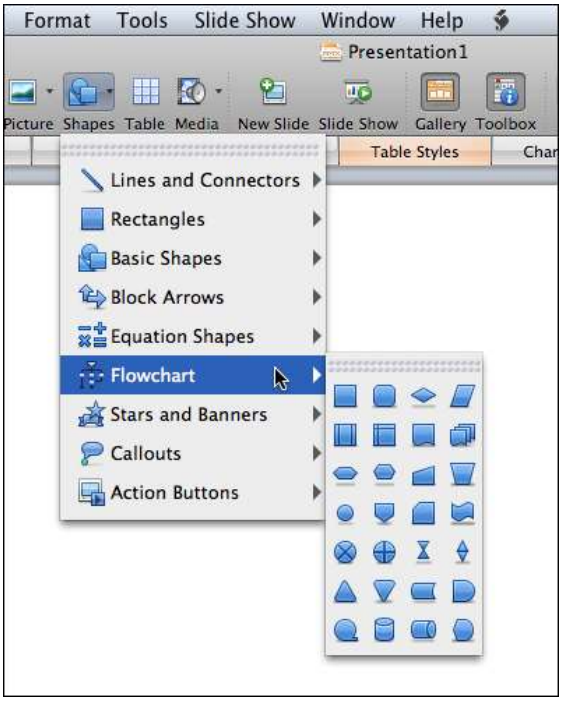


Figure 8-29. *Microsoft PowerPoint*

Figure 8-30. *iWeb "Theme" Dropdown Chooser*

In other libraries

http://quince.infragistics.com/Patterns/Drop%20Down%20Chooser.aspx

You could also look online for specific types of Dropdown Choosers, such as color pickers, date pickers or calendars, font pickers, or numeric sliders.

List Builder

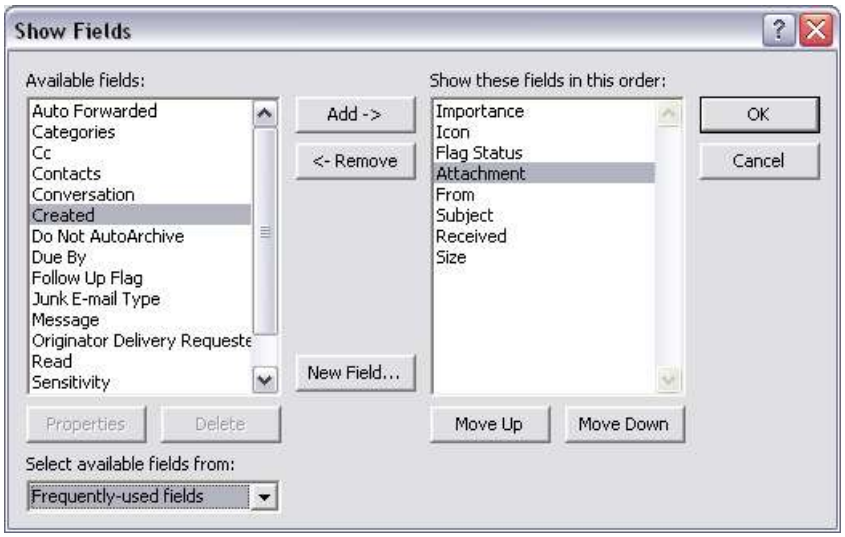


Figure 8-31. *A dialog box from Microsoft Outlook*

What

Show both the "source" and the "destination" lists on the same page; let the user move items between them, via buttons or drag-and-drop.

Use when

You're asking the user to create a list of items by choosing them from another list. The source list may be long—too long to easily show as a set of checkboxes, for instance.

Why

The key to this pattern is to show both lists on the same page. The user can see what's what—she doesn't have to jump to and from a modal chooser dialog box, for instance.

A simpler alternative to List Builder might be a single list of checkbox items. Both solve the "select a subset" problem. But if you have a very large source list (such as an entire filesystem), a list of checkboxes doesn't scale—the user can't easily see what's been checked off, and thus may not get a clear picture of what she selected. She has to keep scrolling up and down to see it all.

How

Put the source list and the destination list next to each other, either left to right or top to bottom. Between the two lists, put Add and Remove buttons (unless your users find drag-and-drop to be obvious, not requiring explanation). You could label the buttons with words, arrows, or both.

This pattern provides room for other buttons, too. If the destination list is ordered, use Move Up and Move Down buttons, as shown in Figure 8-31. (They could have arrow icons too, instead of or in addition to the words.)

Depending on what kind of items you deal with, you could either move the items literally from the source to the destination—so the source list "loses" the item—or maintain a source list that doesn't change. A listing of files in a filesystem shouldn't change; users would find it bizarre if it did, since they see such a list as a model of the underlying filesystem, and the files aren't actually deleted. But the list of "Available fields" in the Outlook example in Figure 8-31 does lose the items. That's a judgment call.

Give the lists multiple-selection semantics instead of single-selection, so users can move large numbers of items from list to list.

Examples

Most modern implementations of this pattern depend upon drag-and-drop to move items between areas; if those items are visual, all the better. Flickr, shown in Figure 8-32, demonstrates a more contemporary approach to a List Builder: you can drag items from a potentially long list of source images into a "batch" group in order to perform operations on all batched images at once. Large text tells the user what to do at critical moments in the interaction, such as starting a new batch or removing an image from the batch.

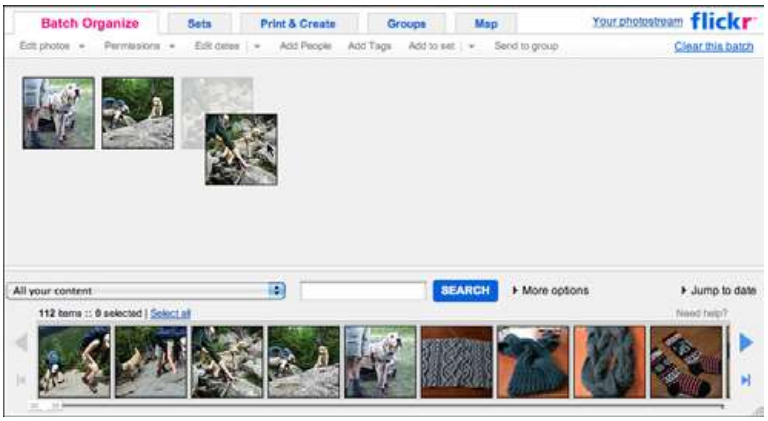


Figure 8-32. *Flickr*

In other libraries

http://www.welie.com/patterns/showPattern.php?patternID=parts-selector

Good Defaults

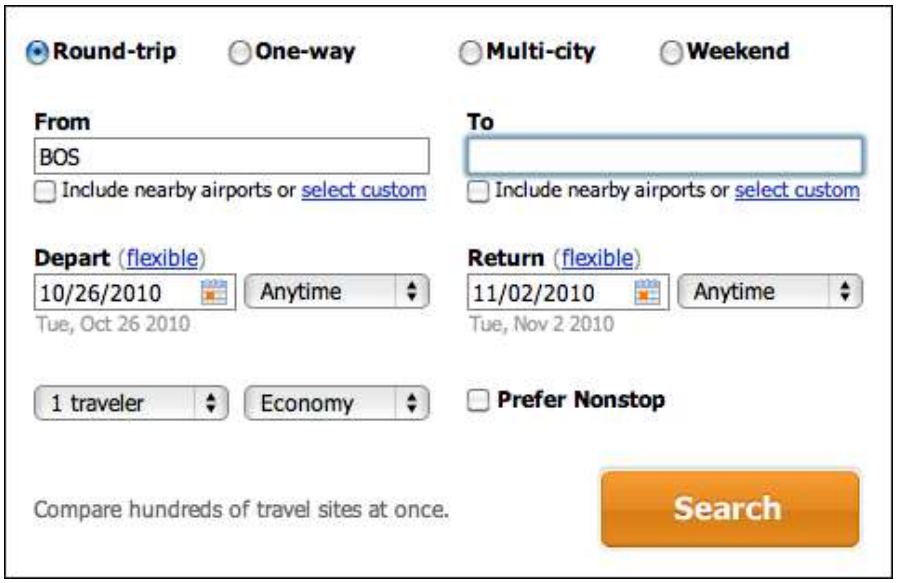


Figure 8-33. *Kayak*

What

Wherever appropriate, prefill form fields with your best guesses at the values the user wants.

Use when

Your UI asks the user any questions requiring form-like input (such as text fields or radio buttons), and you want to reduce the amount of work that users have to do. Perhaps most users will answer in a certain way, or the user has already provided enough contextual information for the UI to make an accurate guess. For technical or semirelevant questions, maybe he can't be expected to know or care about the answer, and "whatever the system decides" is OK.

But supplying defaults is not always wise when answers might be sensitive or politically charged, such as passwords, gender, or citizenship. Making assumptions like that, or prefilling fields with data you should be careful with, can make users uncomfortable or angry. (And for the love of all that is good in the world, don't leave "Please send me advertising email" checkboxes checked by default!)

Why

By providing reasonable default answers to questions, you save the users work. It's really that simple. You spare the user the effort of thinking about, or typing, the answer. Filling in forms is never fun, but if having default answers provided halves the time it takes the user to work through the form, he'll be grateful.

Even if the default isn't what the user wants, at least you offered an example of what kind of answer is asked for. That alone can save him a few seconds of thought—or, worse, an error message.

Sometimes you may run into an unintended consequence of Good Defaults. If a user can skip over a field, that question may not "register" mentally with him. He may forget that it was asked; he may not understand the implications of the question, or of the default value. The act of typing an answer, selecting a value, or clicking a button forces the user to address the issue consciously, and that can be important if you want the user to learn the application effectively.

How

Prefill the text fields, combo boxes, and other controls with a reasonable default value. You could do this when you show the page to the user for the first time, or you could use the information the user supplies early in the application to dynamically set later default values. (For instance, if someone supplies a U.S. zip code, you can infer the state, country, and municipality from just that number.)

Don't choose a default value just because you think you shouldn't leave any blank controls. Do so only when you're reasonably sure that most users, most of the time, won't change it—otherwise, you will create extra work for everybody. Know your users!

Occasional-use interfaces such as software installers deserve a special note. You should ask users for some technical information, such as the location of the install, in case they want to customize it. But 90% of users probably won't. And they won't care where you install it, either—it's just not important to them. So it's perfectly reasonable to supply a default location.

Examples

Kayak (Figure 8-33) supplies default values when a user begins a search for flights. Most are quite reasonable: a round-trip economy flight with one traveler is common, and the "From" city can be derived from either the user's geographic location or the user's previous searches. (The departure and arrival dates seem arbitrary, however.) The effect of having all these defaults is that the user spends less time thinking about those parts of the form, and she gets a quicker path to her immediate goal—the search results.

When an image canvas is resized in Photoshop, the dialog box shown in Figure 8-34 appears. The original image was 476 × 306, as shown. These dimensions become the default Width and Height, which is very convenient for several use cases. If I want to put a thin frame around the image, I can start with the existing dimensions and increase them by just two pixels each; if I want to make the image canvas wider but not taller, I only need to change the Width field; or I could just click OK now and nothing changes.

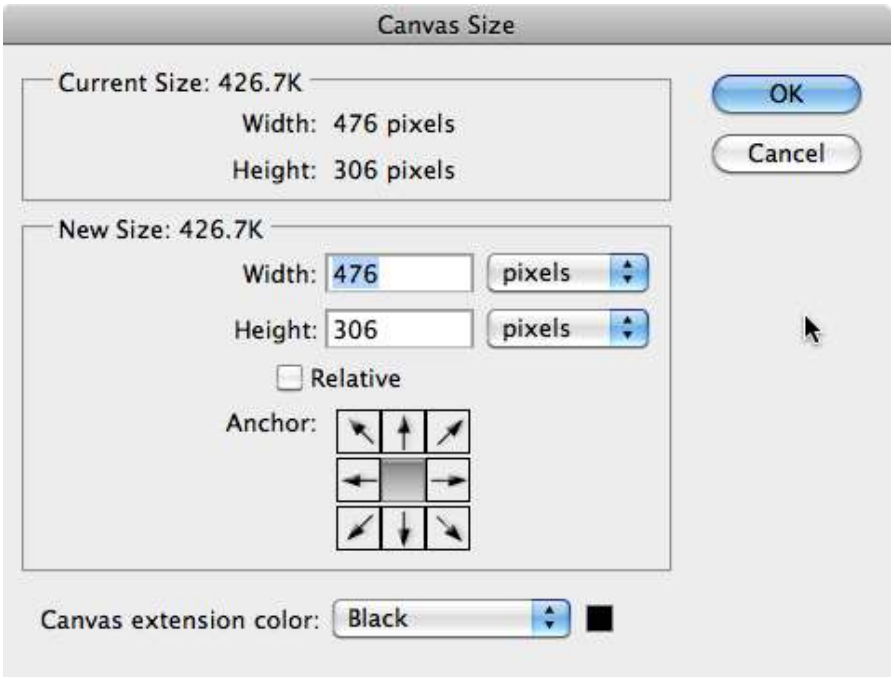


Figure 8-34. *A dialog from Photoshop*

In other libraries

http://patternry.com/p=good-defaults/

http://ui-patterns.com/patterns/GoodDefaults

Same-Page Error Messages

Figure 8-35. *Netflix registration box*

What

Place form error messages directly on the page with the form itself; mark the top of the page with an error message, and if possible, put indicators next to the originating controls.

Use when

Users might enter form information that somehow isn't acceptable. They may skip required fields, enter numbers that cannot be parsed, or type invalid email addresses, for instance. You want to encourage them to try again. You want to point out typos before they become a problem, and help puzzled users understand what is asked for.

Why

Traditionally, applications have reported error messages to users via modal dialog boxes. Those messages could be very helpful, pointing out what the problem was and how you could fix it. The problem is that you had to click away the modal dialog box to fix the error. And with the dialog box gone, you couldn't read the error message anymore! (Maybe you were supposed to memorize the message.)

Then, when web forms came along, error messages often were reported on a separately loaded page, shown after you clicked the Submit button. Again, you can read the message, but you have to click the Back button to fix the problem; once you do that, the message is gone. Then you need to scan the form to find the field with the error, which takes effort and is error-prone.

Most web forms now place the error message on the form itself. By keeping both messages and controls together on the same page, you allow the user to read the message and make the form corrections easily, with no jumping around or error-prone memorization.

Even better, some web forms put error messages physically next to the controls where the errors were made. Now the user can see at a glance where the problems were—no need to hunt down the offending field based just on its name—and the instructions for fixing it are right there, easily visible.

How

First, design the form to prevent certain kinds of errors. Use drop downs instead of open text fields, if the choices are limited and not easy to type. For text fields, offer Input Hints, Input Prompts, Forgiving Format, Autocompletion, and Good Defaults to support text entry. Clearly mark all the required fields as required (with asterisks), and don't ask for too many required fields in the first place.

When errors do happen, you should show some kind of error message on top of the form, even if you put the detailed messages next to the controls. The top is the first thing people see. (It's also good for visually impaired users—the top of the form is read to them first, so they know immediately that the form has an error.) Put an attention-getting graphic there, and use text that's stronger than the body text: make it red and bold, for instance.

Now mark the form fields that caused the errors. Put specific messages next to them, if you can—this will require extra space beside, above, or below the fields—but at the least, use color and/or a small graphic to mark the field, as shown in Figure 8-35.Users commonly associate red with errors in this context. Use it freely, but since so many people are colorblind with respect to red, use other cues, too: language, bold text (not huge), and graphics.

If you're designing for the Web or some other client/server system, try to do as much validation as you can on the client side. It's much quicker. Put the error messages on the page that's already loaded, if possible, to avoid a page-load wait.

A tutorial on error-message writing is beyond the scope of this pattern, but here are some quick guidelines:

- Make them short, but detailed enough to explain both which field it is and what went wrong: "You haven't given us your address" versus "Not enough information."
- Use ordinary language, not computerese: "Is that a letter in your zip code?" versus "Numeric validation error."
- Be polite: "Sorry, but something went wrong! Please click 'Go' again" versus "JavaScript Error 693" or "This form contains no data."

Examples

Twitter's and Mint's registration pages (Figures 8-36 and 8-37, respectively) show either an error message or an "OK" message. This can help for short forms.

Figure 8-36. *Twitter registration page*

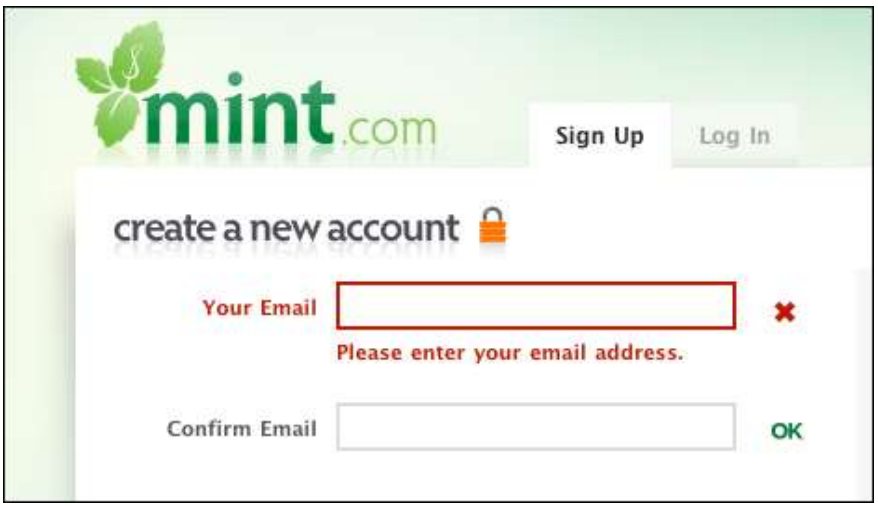


Figure 8-37. *Mint registration page*

Yahoo! uses humor in some of its error messages, while others are straight (see Figure 8-38).

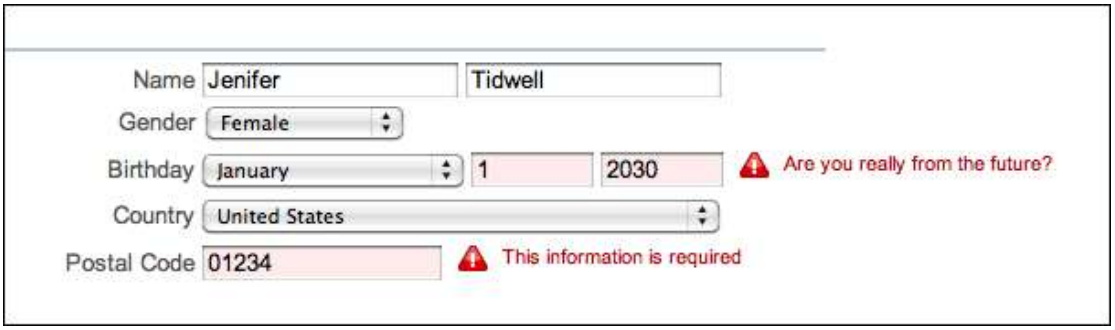


Figure 8-38. *Yahoo! registration page*

When you add a not fully specified item to your cart at Hanna Andersson's site, it uses a gentle message to remind you to fill in missing information, as shown in Figure 8-39. (The Input Prompt makes it too easy to overlook this field on the form, actually.) Once you do add it, the same space might be used for an additional message of interest. Note also that once the form detects that enough information is present, it puts the Begin Checkout button on the form.

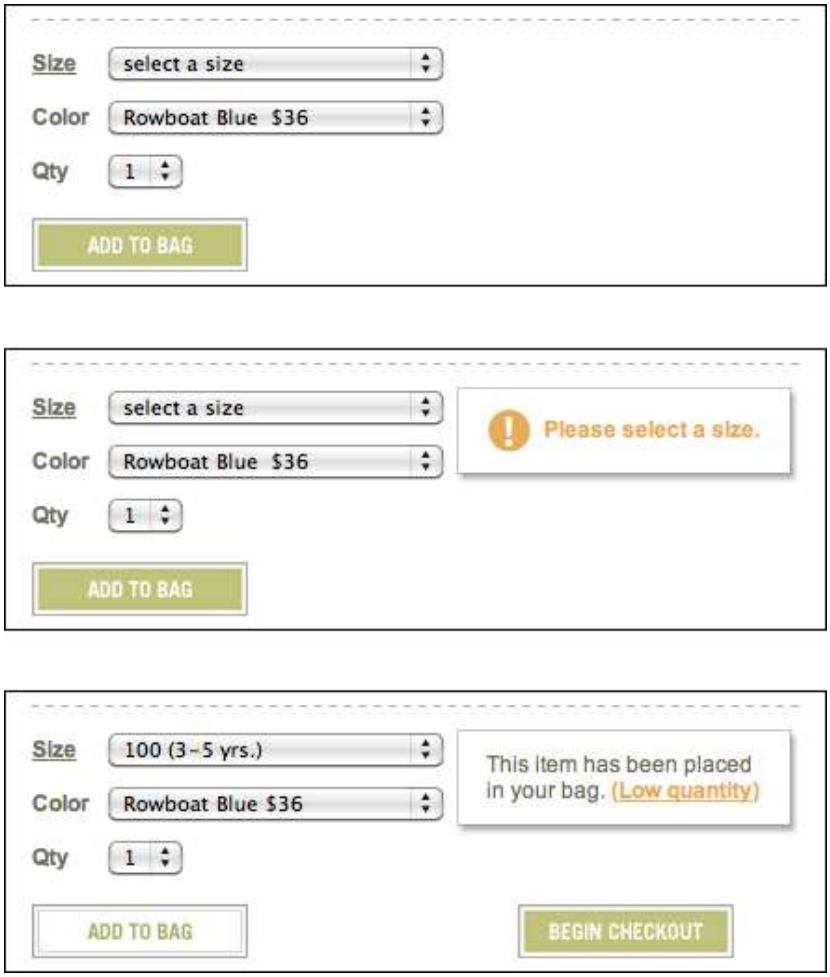


Figure 8-39. *Hanna Andersson's purchase form*

In other libraries

http://ui-patterns.com/patterns/InputFeedback

http://www.welie.com/patterns/showPattern.php?patternID=input-error

These two patterns are named "Input Feedback" and "Input Error Message." You can search for similar variations on the pattern name.