UI Patterns and Techniques

Jenifer Tidwell

Table of Contents

Introduction

There's nothing new here.

If you've done any Web or UI design, or even thought about it much, you should say, "Oh, right, I know what that is" to most of these patterns. But a few of them might be new to you, and some of the familiar ones may not be part of your usual design repertoire.

Each of these patterns (which are more general) and techniques (more specific) are intended to help you solve design problems. They're common problems, and there's no point in reinventing the wheel every time you need, say, a sortable table -- plenty of folks have already done it, and learned how to do it well. Some of that knowledge is written up here, in an easily-digestible format.

By the way, when I say "UI," I mean Web sites, desktop applications, and everything in between (Web forms, Flash, applets, etc.). I believe that over the next few years, Web-based UIs will become more richly interactive than they are now, and the smartest Web designers will use the desktop world's hard-won knowledge of how to design good interactive software. Likewise, desktop applications will gradually look more like Web sites, with better graphic design and more Web-style navigation. I will make no assumptions about how or when they will converge -- they may not, ultimately -- but stylistically, there is some common ground already. Thus, you will see examples from both worlds in here.

These patterns are intended to be read by people who have some knowledge of UI design concepts and terminology: dialogs, selection, combo boxes, navigation bars, whitespace, branding, and so on. It does not identify widely-accepted techniques such as wizards, as you probably already know what they are.

If you're running short on ideas, or hung up on a difficult design quandary, read over these and see if any of them are applicable. And don't take them as the gospel truth, either -- what matters is whether your design works for your users.

If these are useful to you, please tell me. If not, or if you have anything to add, tell me that too.

Jenifer Tidwell
jtidwell @ alum.mit.edu
May, 2002

About Patterns

In essence, patterns are structural and behavioral features that improve the "habitability" of something -- a user interface, a Web site, an object-oriented program, or even a building. They make things more usable, easier to understand, or more beautiful; they make tools more ready-to-hand.

As such, patterns can be a description of "best practices" within a given design domain. They capture common solutions to design tensions (usually called "forces" in pattern literature) and thus, by definition, are not novel. They aren't off-the-shelf components; each implementation of a pattern is a little different from every other. They aren't simple rules or heuristics either. And they won't walk you through an entire set of design

decisions -- if you're looking for a complete step-by-step description of how to design a UI, this isn't the place!

In this work, patterns are described literally as solutions to design problems, because part of their value lies in the way they resolve tensions in various design contexts. For instance, a UI designer who needs to pack a lot of stuff into a too-small space can use a *Card Stack*. All that remains to the designer is to work on the information architecture -- how to split up the content into pieces, what to name them, etc. -- and what exactly the Card Stack will look like when it's done. Tabs? A left-hand-side list or tree? That's up to the designer's judgement.

Some very complete sets of patterns make up a "pattern language." These are a bit like visual languages, in that they cover the entire vocabulary of elements used in a design (though pattern languages are more abstract and behavioral; visual languages talk about shapes, colors, fonts, etc.). This set isn't nearly so complete, and it contains techniques that don't quite qualify as patterns. However, it is concise enough to be manageable and useful.

How to Use These Patterns

- If you don't have years of design experience already, a set of patterns may serve as a learning tool. You may want to read over it to get ideas, or refer back to specific patterns as the need arises.
- Each pattern in this collection has at least one example. Some have many; they might be useful as a sourcebook. You may find wisdom in the examples that is missing in the text of the pattern.
- If you talk to users, engineers, or managers about UI design, or write specifications, then you could use the pattern names as a way of communicating and discussing ideas. This is another well-known benefit of pattern languages.

Ultimately, you should be able leave a reference like this behind. Experienced designers will have internalized these ideas to the point at which they don't notice they're using them anymore; the patterns become second nature.

Other Pattern Collections

The text that started it all was written about physical buildings, not software. Christopher Alexander's *A Pattern Language*, and its companion book *The Timeless Way of Building*, established the concept of patterns and described a 250-pattern multilayered pattern language. It is often considered the gold standard for a pattern language because of its completeness, its rich interconnectedness, and its grounding in the human response to our built world.

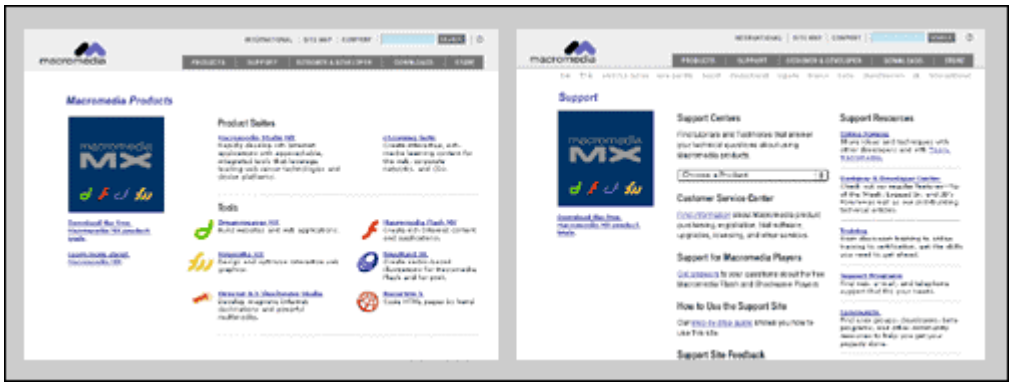
In the mid-1990s, the practice of commercial software architecture was profoundly changed by the publication of *Design Patterns*,by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. This is a collection of patterns describing object-oriented "micro-architectures." If you have a background in software engineering, this is the book that probably introduced you to the idea of patterns. Many other books about software patterns have been written since. Software patterns such as these do make software more habitable -- for those who write the software, not those who use it!

The first substantial set of user-interface patterns was the predecessor of this patterns collection, Common Ground. Many other collections and languages followed, notably

Martijn van Welie's Interaction Design Patterns, and Jan Borchers's book A Pattern Approach to Interaction Design. Very recently, a full-fledged Web site pattern language was published, called The Design of Sites. I highly recommend it, especially if you're designing traditional Web sites. If you're building applets or desktop applications, or if you're pushing the boundaries in either domain, take a look at all of these; you might find inspiration in any of them.

Whole UI

Visual Framework



From http://macromedia.com

Use when: You're building a Web site with several pages, or a UI with several windows -- in other words, **almost always**. You want it to "hang together" and look like one thing, deliberately designed; you want it to be easy to use and navigate.

Why: When a UI uses consistent color, font, and layout, and when titles and navigational aids -- "signposts" -- are in the same place every time, users **know where they are** and where to find things. They don't have to figure out a new layout each time they switch context from one page or window to another.

Ever seen a book in which the page numbers and headings were in a different place on each page?

A strong visual framework, repeated on each page, helps the **page content** stand out more. That which is constant fades into the background of the user's awareness; that which changes is noticed. Furthermore, if you add enough character to the design of the visual framework, this helps with the **branding** of your Web site or product -- the pages become recognizable as yours.

How: Draw up an overall look-and-feel for each page or dialog in the thing you're building. Home pages and main windows are "special" and are usually laid out a little differently from subsidiary pages, but they should still share certain characteristics with the rest of the artifact.

- **Color**: backgrounds, text colors, accent color, etc.
- **Fonts**: for titles, subtitles, ordinary text, minor text.

- **Writing style** and grammar: titles, names, content, short descriptions, any long blocks of text, anything that uses language.

All other pages or windows should also share the following, as appropriate:

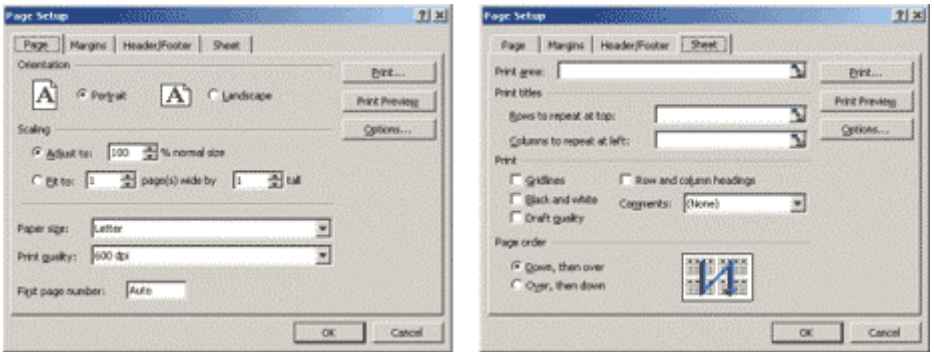
- **"You are here" signposts**: titles, logos, breadcrumb trails, Card Stack indexes such as tabs or link columns.
- **Navigational devices**: sets of standard links, OK/Cancel buttons, back buttons, "quit" or "exit" buttons, *Toplevel Navigation.*
- Techniques used to define *Titled Sections.*
- **Spacing and alignment**: page margins, line spacing, the gaps between labels and their associated controls, text and label justification.
- **Overall layout "grid:"** the placement of things on the page, in columns and/or rows, taking into account the margins and spacing issues listed above.

Implementation of a visual framework should make you separate stylistic aspects of the UI from the content. This isn't a bad thing. If you define the framework in only one place -- a CSS stylesheet, a Java class, etc. -- it lets you change the framework independently from the content, which means you can tweak it and get it right more easily.

Examples:

From http://google.com

Web page designers know how to do this well. Google's pages are simple and unfussy, but very, very recognizable. All the signposts are clear and consistent -- the logo, the page title ("Image Search," "Groups"), the tabs, the other links -- and the all-important search field is *always* in the same place! Note also the use of *Color-Coded Divisions* to distinguish one section from another.



From Excel for Windows

The Windows look-and-feel (or Mac OS) helps to implement a visual framework, since colors, fonts, and controls are fairly standard. These Excel screenshots come from the same dialog, but they illustrate the concept well -- note the consistent location of action buttons in the upper right, the margins and alignment, and the use of label/separator pairs to delimit Titled Sections.

Toplevel Navigation

From http://newscientist.com

Use when: You're building a large Web site, or a complex application with several separate divisions or tools. In either case, users are likely to want to move from one division to another.

Why: On the Web, a toplevel navigation bar is a well-established convention, so users will expect it. More importantly, though, a set of links or buttons that reflects the UI's highest-order structure **makes that structure visible** to users. It gives them an overview of the UI, and helps them find what they need in it (if the naming scheme makes sense, anyway). It also facilitates **exploration** and easy movement, by putting each division only one click away from anyplace else.

You can add to the utility of a toplevel navigation panel by making it show what division the user is currently in. Thus it becomes a **signpost** as well as a navigational device.

How: The most important thing is to use a sensible organizational structure to start with. Keep the number of divisions down to something you can display in the space available, and name the divisions well -- not too many words, meaningful to the user, and following whatever conventions are appropriate ("About Us," "Products," etc.).

As for the toplevel navigation itself, design a single panel that **looks the same** -- and goes into the same place -- on each page where it appears. On the Web,

that should be every page. On a desktop UI, there are far fewer conventional uses of such a thing, but it should probably go into every major application window (not necessarily every dialog). A good toplevel navigation panel is one component of a well-designed *Visual Framework.*

To show where the user is now, simply make the link for the current division look different from the others. Use a contrasting color, perhaps, or an inobtrusive graphic like an arrow.

One design issue that you may run into, especially on Web pages, is how to present this kind of navigational device along with other sets of links. They ought to be distinct from each other. Users may look to the top of the page for the toplevel navigation; that leaves the lefthand and righthand sides for other links, or you could put them in the *Center Stage* and/or content area itself. You could also use two very different sets of affordances -- simple clickable text for the toplevel navigation, and tabs for more "local" things, for instance.

As with Center Stage, keep in mind that home pages and main windows may require different layouts than other pages in the UI. If getting to the different sections of the UI is one of the purposes of the home page or opening window, then toplevel navigation there may need to be more prominent than everywhere else, and you might want to flesh it out with details or sublinks.

Finally, understand that not every user will use, or even notice, a navigational device like this. It's a common misconception among engineers and designers that users will logically look for the overview first, then decide where to go. They won't. They often don't care to know how the site or UI is organized, but simply follow the nearest and most obvious signposts until they find what they need. It's analogous to someone looking for the restrooms in a museum -- they probably won't bother reading a map if there are signs or architectural clues.

Examples:

From http://aiga.org. Note the Flash dropdown menu, which offers more choices in a small space.

From http://economist.com. This navigation bar contains other tools in addition to toplevel links.

Color-Coded Divisions

From http://johncoltrane.com

Use when: You're building a **large UI** with many pages or windows, which can be organized into divisions (chapters, sections, sub-applications, etc.). You might be using a *Visual Framework* to unify them visually. But you also want each division to have a **distinctive look**.

Why: This is an example of a **"signpost"** -- something that gives the user a clue where they are. It does so with some subtlety; colors work visually instead of verbally, and it's not even something that users will necessarily notice immediately (though it's hard to miss in the vivid Coltrane example above).

But once users are attuned to the color schemes, they can use them. Even before then, they'll know when they've left one section for another, if they notice that the color scheme just changed.

So color-coding works to distinguish one section from another; it makes the boundaries clear. It's easier for users to mentally map out smaller chunks of a navigational space, e.g. one division, than the whole space at once -- you should do this with a large UI in any case, whether you use color-coding or not.

Creative uses of different colors could also make your UI **look nicer** and less boring. It might even contribute to the branding of the UI -- see the Apple example below.

How: Pick one of the UI colors and change it from division to division. Usually, the background color is too much -- the Coltrane example only works because the visual framework is so strong and distinctive. Most designs work better with a trim color, like a border, or the background of a small amount of text.

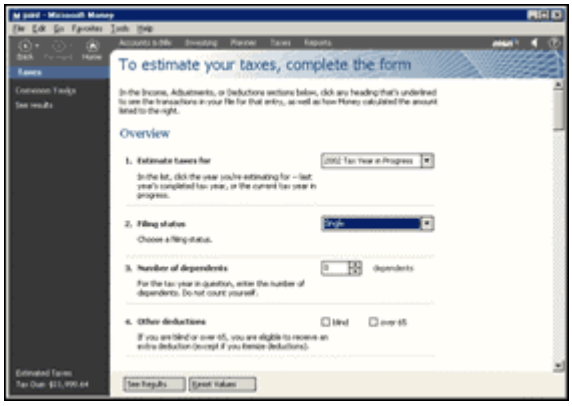
Examples:

From http://apple.com

Apple provides us with a more typical example of color-coding. Look at the top of each screenshot. The tab and the bar below it change color (and texture!) to match the content -- teal for QuickTime, leopard spots for OS X "Jaguar," etc. The effect is subdued but noticeable. It contributes to both usability and branding, while not detracting from the unity of the overall site. (Note that the tabs are the *Toplevel Navigation* in this Web site, while the secondary navigation links live on the colored bar.)

Page Layout

Center Stage



From Microsoft Money

Use when: The page's primary job is to show coherent information to the user, or enable them to perform certain tasks -- in other words, any **information-centered** or **task-centered** interface. Tables and spreadsheets, forms, Web pages containing textual content, and graphical editors all qualify.

Why: The user's eye should be guided immediately to the start of the most-important information (or task), rather than wandering over the page in confusion. An unambiguous central entity "anchors" their attention. Just as the lead sentence in a news article establishes the subject matter and purpose of the article, so the entity in center stage establishes the purpose of the UI.

Once that's done, then the user will assess the stuff in the periphery in terms of how they relate to what's in the center. This is easier for the user than repeatedly scanning the page, trying to figure it out -- what comes first, what's second, how this relates to that, etc.

How: Establish a **visual hierarchy** with the "center stage" dominating everything else. Consider these factors, though none of them are absolutely required:

- **Size.** The center-stage content should be at least twice as wide as whatever's in its side margins, and twice as tall as its top and bottom margins. (The user may change its size in some UIs, but this is how it should be when the user first sees it.)
- **Color.** Use a color that contrasts with the stuff in the margins. In desktop UIs, white works well against Windows gray, especially for tables and trees. As it happens, white often works in Web pages too, since ads and navigation bars usually use other colors as their backgrounds; also, Web users have been trained to look for the plain text on a white background.
- **Headlines.** In the above example, the eye is drawn to the big text at the top of the page. That happens in print media too, of course. See *Titled Sections* for more details.
- **Context.** What does the user expect to see when they open up the page? A graphic editor? A long text article? A map? A file system tree? Work

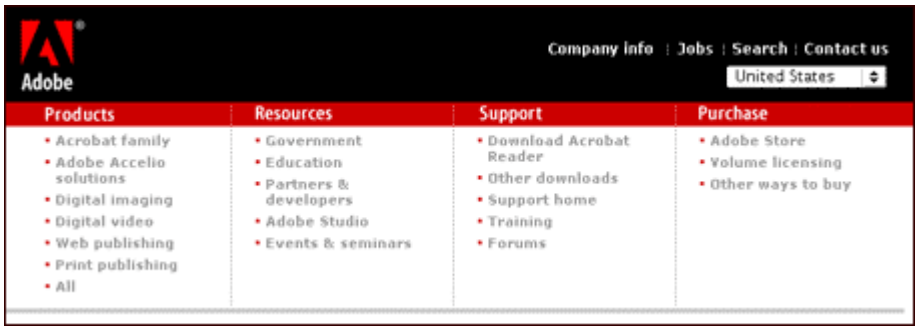
with their preconceptions; put that in center stage and make it recognizable. The user will look for it -- this trumps all other rules about visual perception. (But it doesn't mean you can frustrate the user by hiding what they're looking for! Some Web sites put their main content so far down the page that it's "below the fold" in short windows, requiring the user to scroll down to find it. That's just sadistic.)

Notice that one traditional layout factor was not mentioned: position. It doesn't much matter where you put the center stage -- top, left, right, bottom, center, all of them can be made to work. Keep in mind that well-established genres have conventions about what goes into which margins, e.g. toolbars on top of graphic editors, or navigation bars on the left sides of Web pages. Be creative, but with your eyes open.

If you're in doubt, take a screenshot of the layout, shrink it down, blur it, and ask someone where on the page they think the main content should start.

The name of this pattern came from a paper authored by P. R. Warren and M. Viljoen.

Titled Sections



From http://www.adobe.com

Use when: There's more than a small handful of controls or text fragments on the page. In other words, **almost always** -- unless the content of your UI is very small, or is conceptually one thing, like a textual or visual narrative.

Why: Well-defined and well-named sections **structure the content** into easily-digestible chunks, each of which is now understandable at a glance. It makes the information architecture obvious.

When the user sees a page sectioned neatly into chunks like this, their **eye is guided along the page** more comfortably. The human visual system always looks for bigger patterns, whether they exist or not. So put them in deliberately!

How: First, get the information architecture (IA) right -- split up the content into coherent chunks, and give them short, memorable names (one or two words, if possible). Next, choose a presentation:

- For titles, use a **font that stands out** from the rest of the content -- bold, wider, larger point size, stronger color, etc. (Remember that

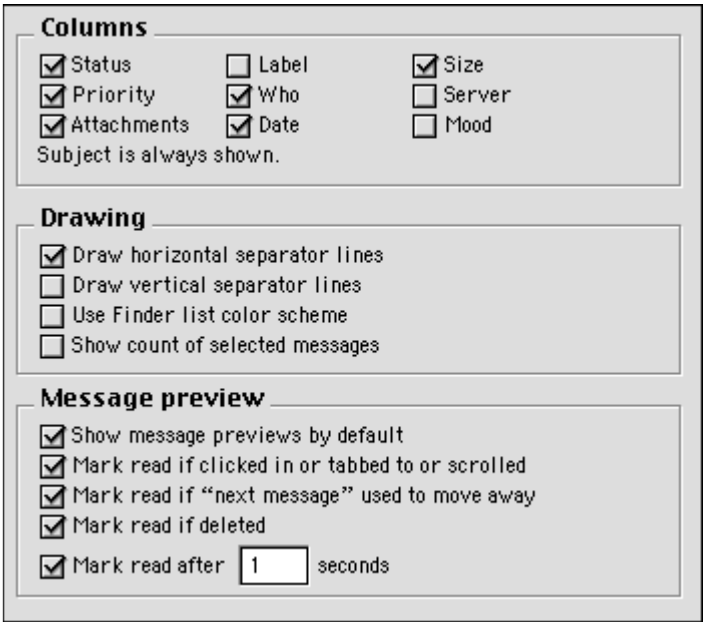
nothing's stronger than black, not even red.)

- Try **reversing the title** against a strip of contrasting color. White on dark can make it look like a Windows title bar.
- Use **whitespace** to separate sections.
- Putting sections on **different background colors** works well on Web pages and "flashy" interfaces, though it's unusual on desktop UIs.
- **Boxes** made from etched, beveled, or raised lines are familiar on desktop UIs. They can get lost -- and just become visual noise -- if they're too big, or too close to each other, or deeply nested. It can be done well when combined with the title; see the examples.

Basically what you're doing here is building a **visual hierarchy** on the screen. This is a concept from graphic design. Grouping, fonts, and judicious use of whitespace can take you a long way.

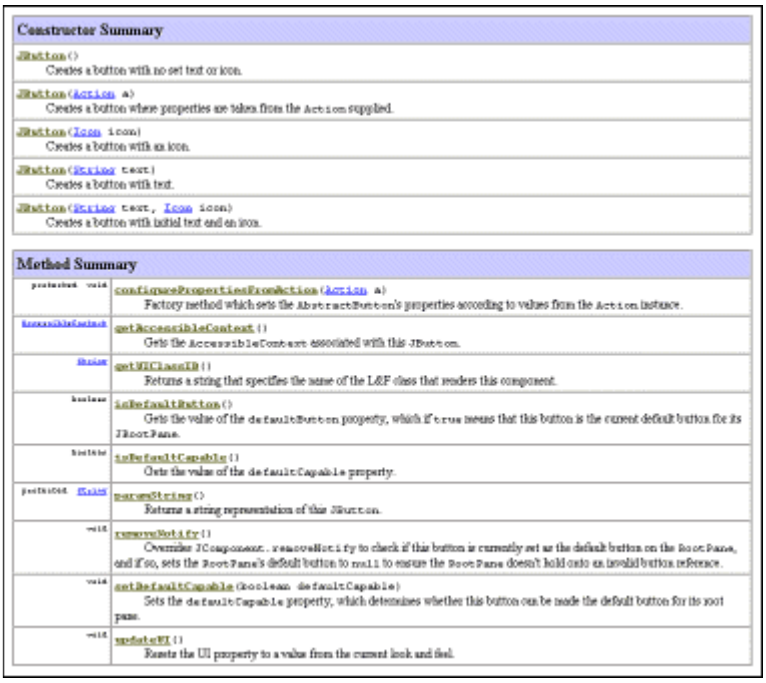
If there's still too much stuff on one page, try *Card Stack, Overview Plus Detail,* or *Extras On Demand* to manage it all. You can combine some of these patterns with Titled Sections, too.

Examples:



From Eudora for Mac OS 9

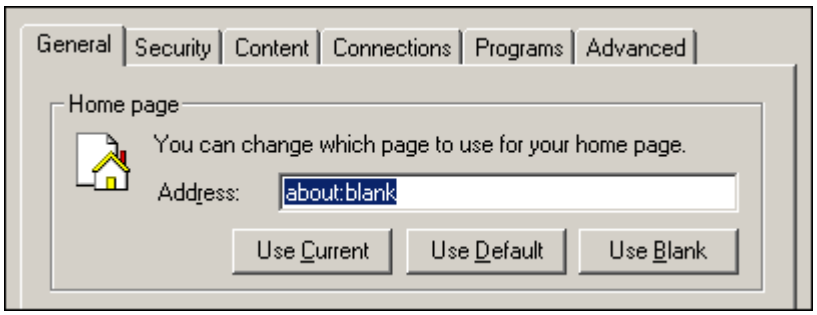
A typical usage in desktop applications. In this example, the boxes look good around the grids of checkboxes, the bold titles stand out clearly, and there is sufficient white space between the sections to give them visual "breathing room." (In fact, this example would work even if the boxes were erased, though it would look a little odd.)



From a Javadoc HTML page

This screenshot came from a long page full of programmer-level Java documentation. Each section is labeled with the blue bars, which are very easy to find and read as the user scrolls rapidly down the page.

Card Stack



From Internet Explorer for Windows

Use when: There's too **much stuff** on the page. A lot of controls or texts are spread across the UI, without benefit of a very rigid structure (like a Property Sheet); the user's attention becomes distracted.

Why: The labeled "cards" **structure the content** into easily-digestible chunks, each of which is now understandable at a glance. It makes the information architecture obvious.

Tabs, especially, are very **familiar** to users. Finally, Card Stacks save **space**.

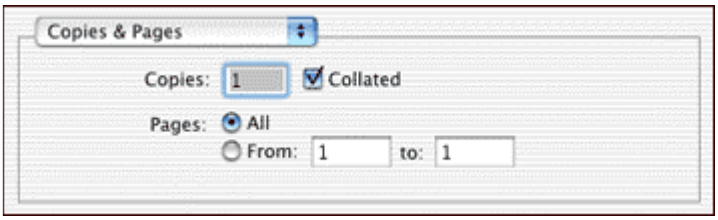
How: First, get the information architecture (IA) right -- split up the content into coherent chunks, and give them short, memorable names (one or

two words, if possible). Then choose a presentation:

- **Tabs** are great, but they usually require 6 or fewer cards. Don't "double-row" them; scroll them horizontally if you must.
- A lefthand **column of names** works well on many Web pages and dialogs. You can fit a lot of cards into one of these. It lets you organize them into a hierarchy, too. (At some point it becomes more like an *Overview Plus Detail;* there's really no clear boundary between them, technically.)
- Some UIs have a **dropdown list** at the top of the page, which takes less space than a link column, but at the cost of clarity. It can work if the containment is very, very obvious; see the example.

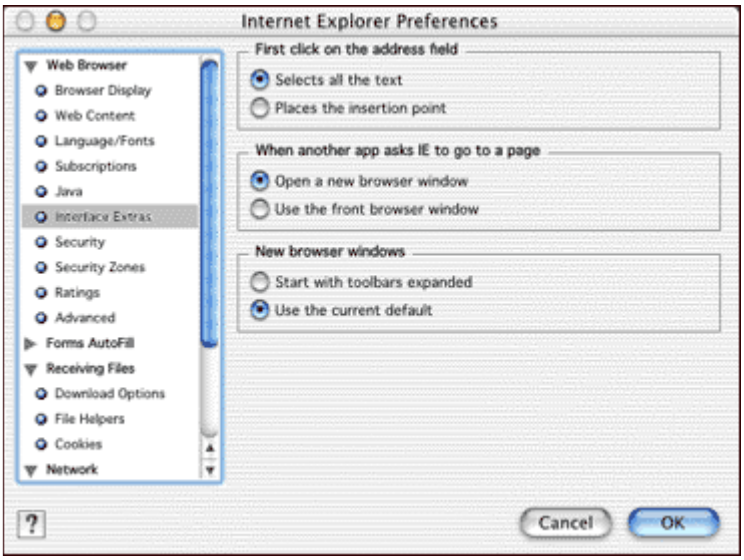
Remember that if you split it up wrong, users will be forced to switch back and forth between cards as they enter information or compare things. Be nice to your users and test the IA.

Examples:



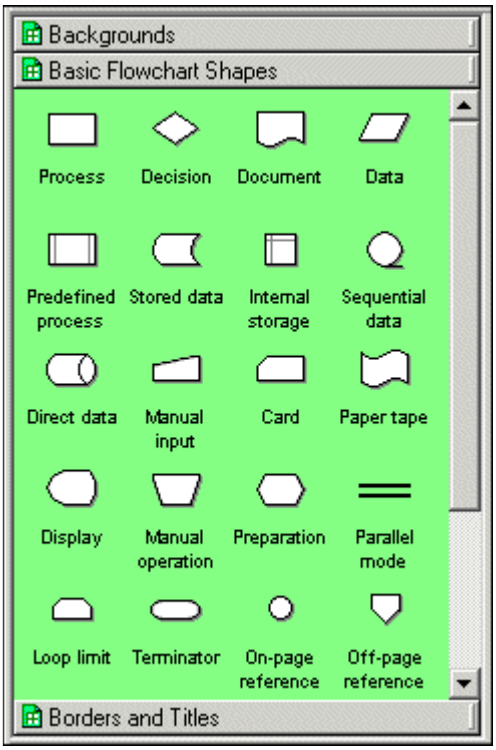
From Internet Explorer for OS X

In Mac OS X, there are many applications that use dropdown lists (here showing "Copies & Pages") in places where tabs might have be used. Dropdowns are space-conserving, and allow for long or numerous page names, but the user can't see what other pages are available until they open the dropdown list. Note the box around the controls; this kind of containment is necessary for a user to understand what the dropdown does. Otherwise, it just looks like another control, not a navigational device.



Also from Internet Explorer

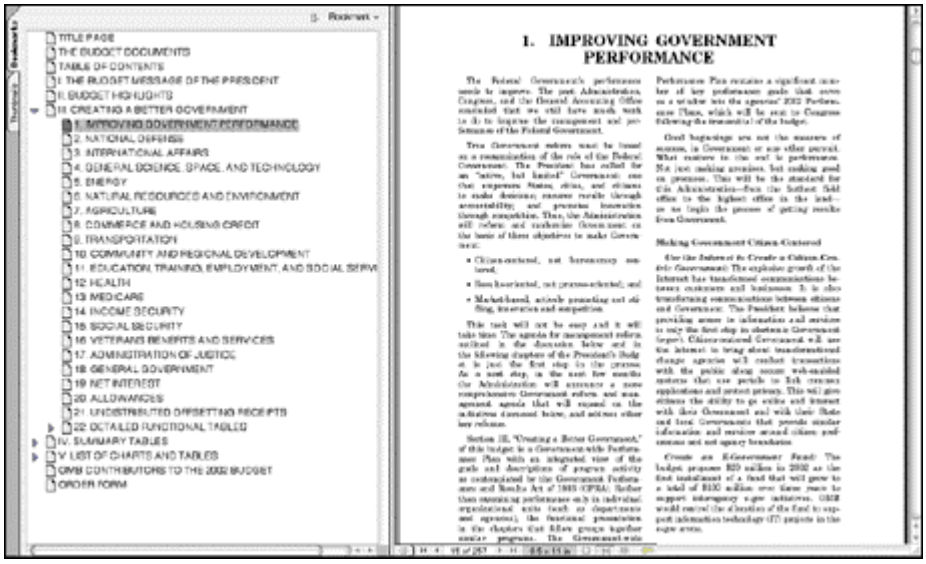
This time the selector is a list on the left. This example walks the line between Card Stack and *Overview Plus Detail.*



From Visio for Windows

The "tab" buttons in this Visio palette are vertically stacked, and they move from top to bottom as the user clicks them, so that the selected page always has its button directly above it. This is an interesting solution for a constrained, vertically-oriented space.

Overview Plus Detail



From Adobe PDF viewer

Use when: The UI presents something -- a composite object, a collection of things, etc. -- that's **too complex or dynamic** to show in just one page. *Titled Sections* doesn't scale; the set of "cards" needed for a *Card Stack* is big or changing, or they won't fit into a simple linear model.

Why: It's an age-old way of dealing with complexity: present a **high-level view** of what's going on, and let the user **"drill down"** from that view into the details as they need to, keeping both levels visible on the page for quick iteration.

An information hierarchy is at work here again, just like Titled Sections and Card Stack. You defeat complexity via divide-and-conquer. Overview Plus Detail structures the content into comprehensible pieces, and you organize them as the content dictates.

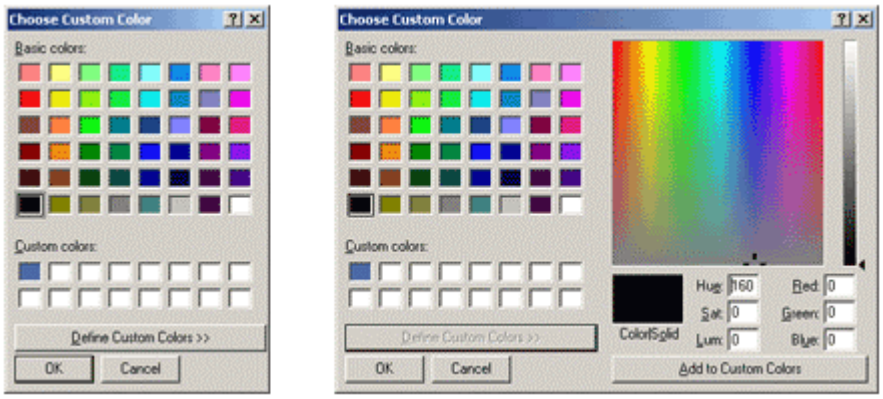
How: The composite object serves as a **selectable "index."** Put it on the lefthand side. When the user selects some part of it, details about that part -- controls, text, data, etc. -- appears on the other side. (You might also choose to put the index object on the top, with the details below.)

What exactly is the "composite object?" It depends. A file system viewer may show the file hierarchy. A map browser may show a map with selectable grid squares. A GUI builder may simply use the layout canvas itself -- selected objects on it show their properties as the "details."

Keeping both halves on the same page or window is key. You can put the details into a separate window, but it's not as effective. You want the user to be able to **browse easily** and fluidly through the UI, without waiting or messing around with windows.

This pattern is easier to illustrate than talk about.

Extras On Demand



From the Windows color dialog

Use when: There's too **much stuff** on the page, but some of it isn't very important. You'd rather have a simpler UI, but you have to put all this content somewhere.

Why: A simple UI is an excellent thing, especially for new users, or users who don't need the full functionality you can provide. **Let the user choose** when to see the entire UI in its full glory -- they're a better judge of that than you are.

If your design makes 80% of the use cases easy, and the remaining 20% are at least possible (with a little work on the user's part), your UI is doing as well as can be expected!

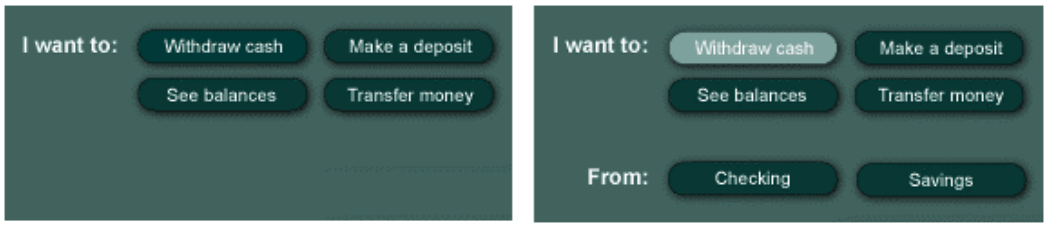
When done right, it can **save space**, too.

How: Ruthlessly **prune the UI** down to its most commonly-used, most important items. Put the remainder into their own page or section. Hide that section by default; on the newly-simplified UI, put a clearly-marked button or link to the remainder, e.g. "More Options."

That section should have another button or other affordance to let the user **close it again**. Remember, most users won't need it most of the time. Just make sure the entrance and exit to this "extras" thing are obvious; test them.

On some dialog boxes, the box literally expands to accommodate the details section, then shrinks down again when the user puts it away. This works well. Another mechanism is found on various desktop UIs: a dropdown for fill color, for instance, contains a "More Fill Colors..." item which brings up a separate dialog box.

Dynamic Progression



From an ATM redesign

Use when: The user should be walked through a complex UI task **step-by-step**, perhaps because the user is computer-naive, or because the task is rarely done. But you don't want something **as constraining as a wizard** -- you'd rather keep the whole interface on one page, for instance.

Why: As the user sees the task unfolding directly in front of them, via a dynamically-changing UI, they can form a correct mental model of the task more quickly and easily. There are none of the awkward **context switches** that separate wizard screens impose.

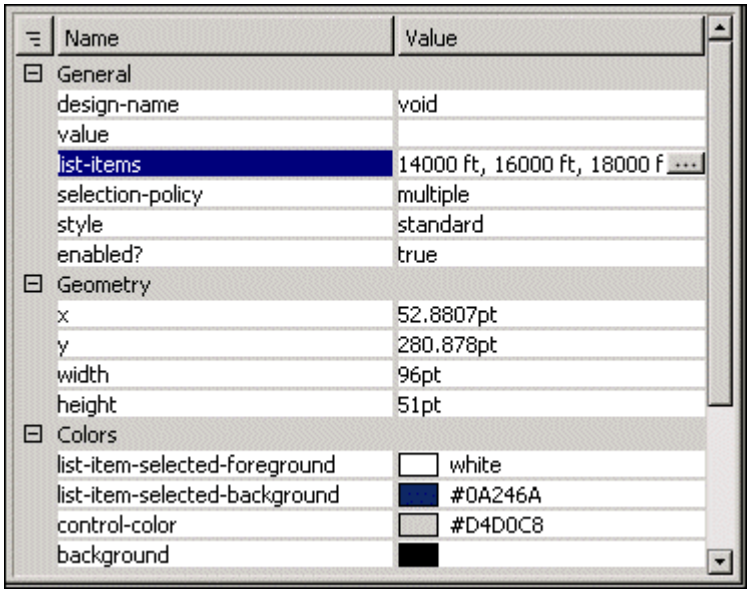
Furthermore, since the UI is kept together on one page, the user can very easily **go back** and change their mind about earlier choices; they immediately see the effect on subsequent steps.

For occasional tasks, this can work better than presenting a complex and interlinked set of controls all at once, because it's **always obvious** what the first step is -- and the next, and the next! The user never has to think too hard.

How: Show the controls for only the first step; when the user's done with that step, show the controls for the next step; etc. "Show" may mean "enable," or literally place on the page. Leave the previous steps' controls visible, so the whole UI is progressively revealed.

In many such step-by-step designs, the choices the user makes at one step alters the rest of the task. For instance, an online order form asks if the billing address is the same as the shipping address. If the user says yes, then the UI doesn't even bother showing entry fields for it. Otherwise, there's one more step in the process.

Property Sheet



From the Curl Surge Lab IDE

Use when: The UI presents an **editable object** to the user -- something built in a graphical editor, or a database record or query, or some domain-specific thing (like a car or a stock purchase). The user will need to change the properties or attributes of the object.

Why: Most users are familiar with the concept of a property sheet -- a list of object properties or settings, set forth in a prescribed order, editable via controls appropriate for the property types (text fields for strings, dropdowns for one-of-many choices, etc.). Simply because they're **conventional,** well-designed property sheets are generally easy to use.

Property sheets can also help the user build a correct **mental model** of the objects in the UI. A property sheet tells the user what the object's properties are, and what values are available for each of them. Especially in applications that mix WYSIWYG editing with programming (such as GUI builders, Web page builders, and time-based scripting and animation tools), property editors thus help the user learn how to use the system.

How: The only commonality here is that the various edit controls are labeled with the names of the properties they edit. When the user is done filling in values, the new property values are written out to the object.

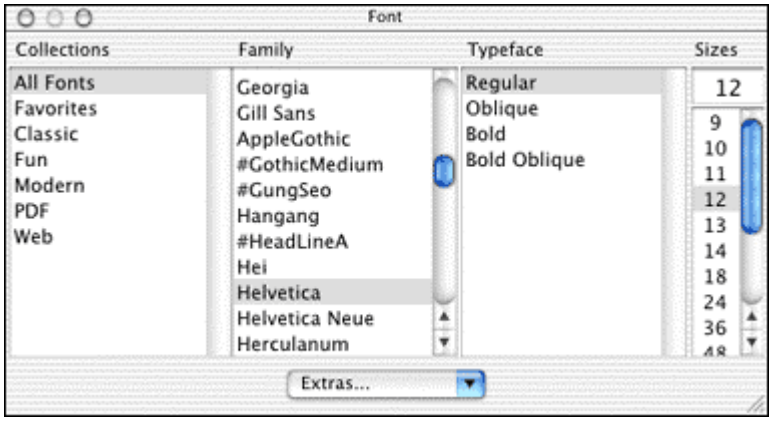
These are the issues that usually come up when designing property sheets:

- **Layout.** Some systems use a two-column table, with controls that appear when the user clicks the values shown in the righthand column. (Visual Basic seems to be the de facto standard for this approach.) Others look more like dialogs than tables -- text fields beside controls. Use your judgement. Tables might be more recognizable as property sheets, but dialogs can be far more flexible in their presentation of the property-editing controls.
- **Property order.** Alphabetical? Categorized? Or an easy-to-read order

that places the most commonly-edited properties at the top? They all have their place. Short property sheets (say, 10 or fewer properties) are usually best with the most common properties listed first. The longer the list gets, the more important it is to categorize them; but if a user is looking for one particular property by name, they may want to sort them alphabetically. As always, it's best to give users a choice. But you're still responsible for picking the right default order.

- **Control choice.** Fifty pages could be written about this. The short version: make sure the property's current value is always there to be seen; choose controls to constrain the input as much as possible, e.g. by using non-editable dropdown lists for one-of-many choices; use built-in fancy editors for specialized types like colors, fonts, and filenames.
- **When to commit** the new property values. Many UIs simply update the object with the new value as soon as the user is done typing or selecting a value. The more dialog-like property sheets may wait until the user deliberately commits the whole thing, e.g. by clicking on an "OK" button. But if your software can deal well with instant update, that gives the user more immediate feedback about the change they just made.
- **"Mixed" values** for multiply-selected objects. Some UIs solve this by showing no value at all, which can be dangerously misleading. Others show it with a sentinel value, like an asterisk "*", or the word "mixed."

Cascading Lists



A font dialog from Mac OS X

Use when: The user needs to navigate a hierarchy that isn't very deep, but might have many items on each level. An outline or "tree control" would work, but the user would be scrolling up and down a lot to see all the items, and they wouldn't get a good overview of the items at higher levels in the hierarchy.

The hierarchy may be a literal one, such as a filesystem, or a conceptual one -- this pattern is often used to let a user navigate and choose items within **categories**, or make a series of interdependent choices, as with the fonts above.

Why: By spreading the hierarchy out across several scrolled lists, you **show more of it at once**. It's that simple. Visibility is good when you're dealing with

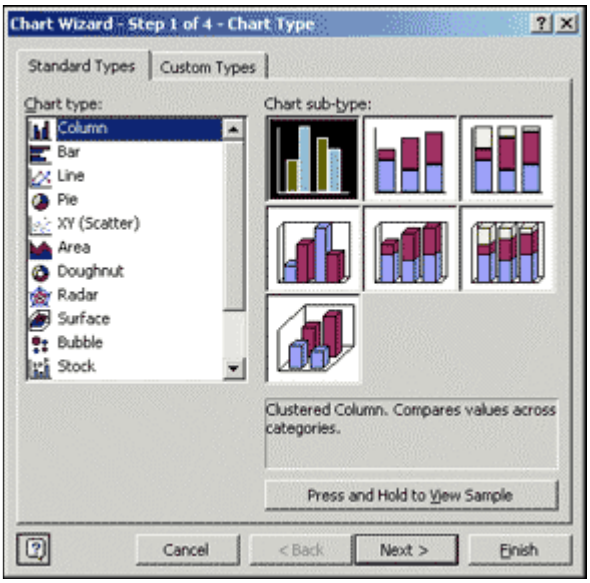
complex information structures. Also, laying the items out in lists organizes them nicely -- a user can more easily keep track of what level they're dealing with than they could with an outline format, since the hierarchy levels are in nice predictable fixed-position lists.

How: Put the first level of the hierarchy in the leftmost list (which should use single-selection semantics). When the user selects an item in it, show that item's children in the next list to the right. Do the same with the child items in this second list; show its selected item's children in the third list. And so on.

Once the user reaches items with no children -- the "leaf" items, as opposed to "branches" -- you might want to show the details of the last-selected item at the far right. In the Mac Finder example below, a representation of an image file is shown; you might instead offer a UI for editing an item, or for reading its content, or whatever is appropriate for your particular application.

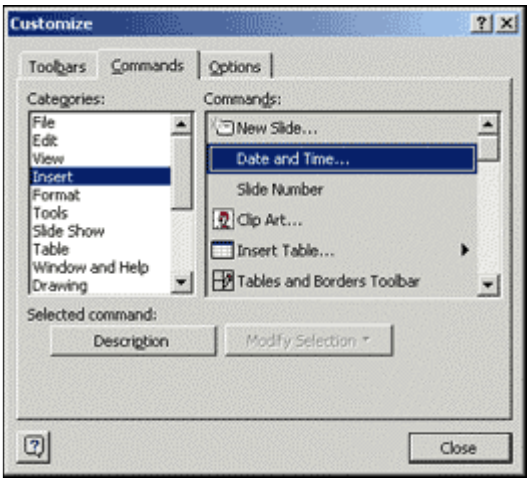
A nice thing about this pattern is that you can easily associate buttons with each list: delete the current item, move up, move down, etc. Many toolkits will let you do this in tree controls via direct manipulation, but for those that don't, this is a viable alternative. See the examples.

Examples:



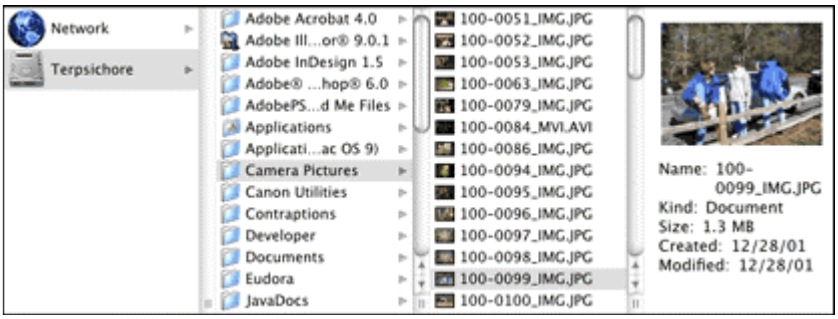
From Excel for Windows

It may not look like one, but this Excel chart-type chooser is a two-level cascading list, expressed with two different visual formats. It uses a category / item information architecture. The user's selection in the "Chart sub-type" list is described with its name and a *Short Description.* (Note also the use of a *Card Stack* and *Illustrated Choices* in this dialog.)



From PowerPoint for Windows

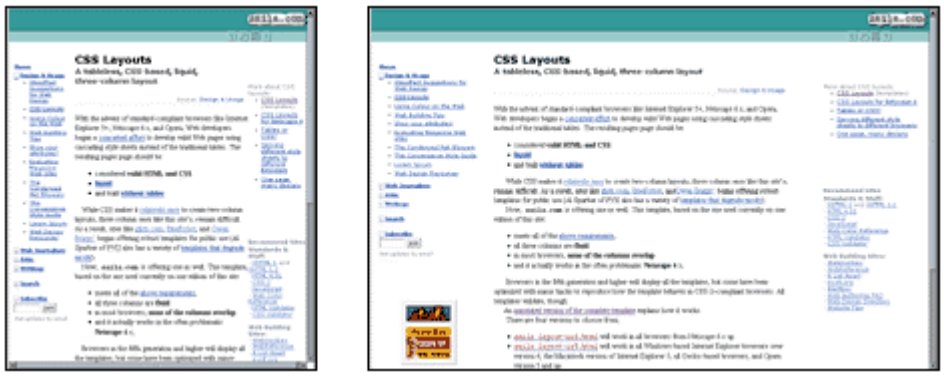
Though simpler than the Excel example, this one uses the same two-level structure, and it uses a similar category / item information architecture.



From the Mac OS X Finder

This is an extreme example, but it shows that the pattern scales well, letting the user drill down into very deep filesystem hierarchies while staying oriented. NextStep originally used this technique in its file finder, circa 1989 or so.

Liquid Layout



From http://www.saila.com

Use when: The user might want **more space** -- or less -- in which to show the content of a window, dialog, or page. This is likely to happen whenever a page contains a lot of text (as in a Web page), or a high-information widget like a table or tree, or a graphic editor. It doesn't work so well when the visual design requires a certain amount of screen real estate, no more, no less.

Why: Unless you're designing a "closed" UI like a kiosk or a full-screen video game, **you can't predict the conditions** under which the user is going to view your UI. Screen size, font preferences, other windows on the screen, the importance of any particular page to the user -- none of this is under your control. How, then, can you decide the one optimal page size for all users?

Giving the user a little control over the layout of the page makes your UI **more flexible** under changing conditions. It may also make the user feel less antagonistic towards the UI, since they can bend it to fit their immediate needs and contexts.

If you need more convincing, consider what happens to a fixed-layout "non-liquid" UI when the language or font size changes. Do columns still line up? Do pages suddenly become too wide, or even clipped at the margins? Generally, pages engineered to work nicely with window size changes will also accommodate these other changes.

How: Make the page contents continuously **"fill" the window as it changes size.** Multiline text should wrap at the right margin, until it becomes four or five inches wide (at which point it becomes too hard to read). Trees, tables, editors, etc. at *Center stage* should enlarge generously while their margins stay compact. If the page has anything form-like on it, horizontal stretching should cause text fields to elongate -- users will appreciate this if they need to type in anything longer than the text field's normal length. Likewise, anything scrolled (lists, tables, etc.) should lengthen, and possibly widen too.

Web pages and similar UIs should allow the body content to fill the new space, while keeping navigational devices and signposts anchored to the top and left margins. Background colors and patterns should always fill the new space, even if the content itself cannot.

What happens when the window gets too small for its content? You could put

scrollbars around it. Otherwise, white space should shrink as necessary; outright clipping may happen when the window gets really tiny, but the most important content hangs in there to the end.

A well-behaved Liquid Layout can be difficult to implement in Web pages, especially if you want to dispense with tables and use straight CSS. It's also hard in Visual Basic and Visual C++. Java, however, provides several layout managers that can be used to implement it.

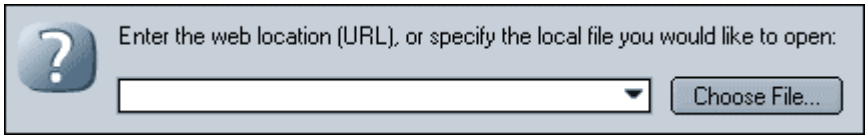
Examples:

From a file-open dialog in Mac OS X.

Mac OS allows you to resize the standard "Open" dialog, which uses a liquid layout. This is good because the user can see as much of the filesystem hierarchy as they want, rather than being constrained to a tiny predetermined space. (The *Cascading Lists* pattern is used in this dialog, too.)

Forms and Input

Chooser Button



From Netscape 6

Use when: The user needs to provide a one-line name or reference for something, such as a file, printer, image, color, font, etc.

Why: It's great if the user can remember the name or path to type, but very often they won't. The UI should let the user go **look for the object** in question, to select it from a broader or more familiar context; this helps prompt the user's memory, and spares them the effort of memorizing (and typing in) the name of the object.

If you only provide a plain text field, the user will probably go use some other tool -- an OS-provided tool, or another application -- to look up the

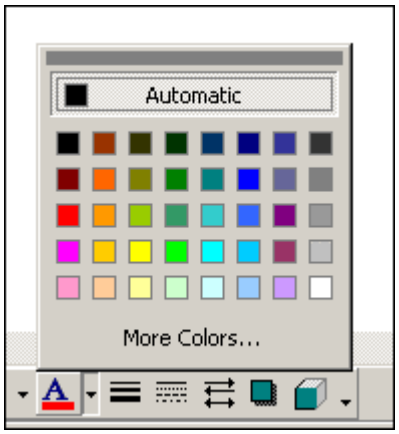
object in question. But that's inconvenient, and by switching to a different tool and context, the user's work flow is broken up.

How: Put a button at the end of the text field that launches a chooser or finder of some sort. That button might have descriptive text, such as "Choose...", "Browse...", "Find...", or a descriptive image, or just "...". See what works for your users. In general, users understand that "..." on a control means it'll prompt them for more input.

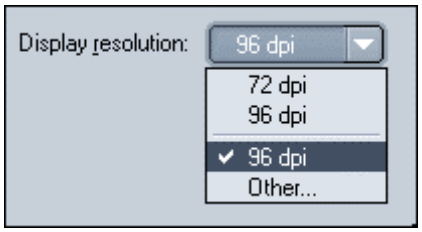
When the user has selected something from the finder (which is usually a modal dialog), close it and put the results into the text field. Now the next time the user wants to specify that object, they know what text to use for it.

The same idea applied to dropdown lists and combo boxes can be implemented with a "More..." or "Custom..." item (for example) at the end of its menu. MS Office does this with many of its dropdowns, especially for color choice.

Examples:

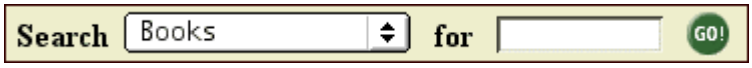


From MS Word



Also from Netscape 6

Fill-in-the-Blanks



From http://amazon.com

Use when: Form input is required to perform some action. A property-sheet-style label/text-field format isn't self-explanatory, but you can **verbally describe** the action to be taken.

Why: 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 required of them.

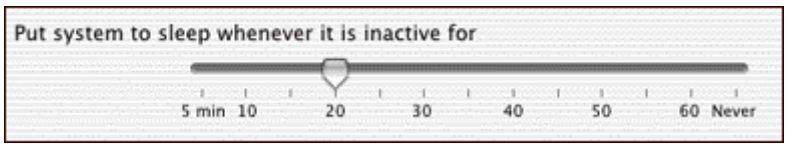
How: Write the sentence or phrase, using all your wordcrafting skills. Put the **controls in place of words.** If you're going to embed the controls in the middle of the phrase, instead of at the end, this works best with text fields, dropdown lists, and combo boxes -- in other words, controls with the same "form factor" as words in the sentence.

Also, for controls in the middle of the phrase, 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 word spacing between them and the surrounding words.

Keep in mind that this pattern makes it very hard to properly localize the UI, since it depends upon word order in a natural language. You may have to rearrange the UI to make it work in a different language.

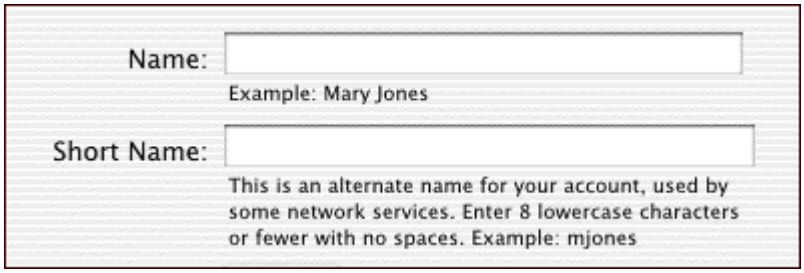
Examples:

From the InConcert Order Browser



From the Mac OS X system preferences

Input Hints



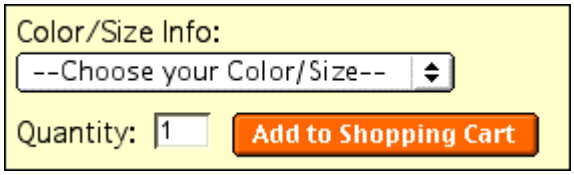
From the Mac OS X system preferences

Use when: The UI presents a text field, but the kind of input it requires isn't patently obvious to all users.

Why: If the UI is **self-explanatory**, the users don't have to guess what to type -- they don't even need to think about it!

How: Write a **short example** or explanatory sentence, and put it below the text field. Two examples conjoined by "or" works fine too. Keep the text small and inconspicuous (though readable); consider using a font two points smaller than the label font.

Input Prompt



From http://rei.com

Use when: The UI presents a text field, dropdown, or combo box for which user input is **required**. Normally you would use a good default value, but you can't in this case -- perhaps there is **no reasonable default**, perhaps for technical or political reasons.

Why: If the UI is **self-explanatory,** the user doesn't have to guess whether they have to deal with this control or not -- the control itself tells them.

Default values are helpful, but as a user scans the UI looking for things they have to do to proceed, they might skip right over a control whose value they really ought to set. (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!) An imperative "Fill me in!" is **likely to be noticed** by the user.

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

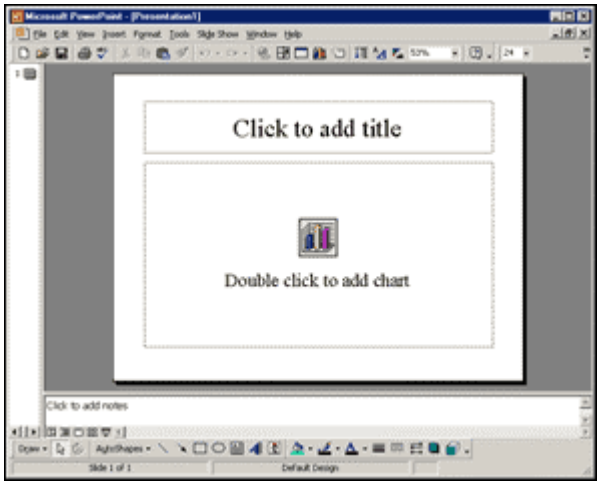
- For a dropdown list, use **Select, Choose,** or **Pick.**
- For a text entry control, use **Type** or **Enter.**

End it with a noun describing what the input is, e.g. "Choose a state," "Type

your message here," "Enter the patient's name." Put this phrase into the control, where the value would normally be. (It shouldn't be a selectable value in a dropdown.)

Since the point of the exercise was to tell the user 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 whatever) that lets the user finish this part of the operation. That way, you won't have to throw an error message at the user.

Examples:



From Powerpoint

Forgiving Format

From http://wunderground.com

Use when: Your UI can accept **input of various kinds** from the user -- different meanings, different formats, etc. -- but you don't want to clutter up the UI with a bunch of separate text fields to "type this OR that OR that OR...".

Why: The user just wants to get something done, not think about complicated either-or choices and fiddly 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 they need, and if it's reasonable, make the software **do the right thing** with it.

This can help **simplify the UI** tremendously. It may even remove the requirement for an Input Hint, though they're often seen together.

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 stuff a user is likely to type in -- maybe you're asking for a date or time, and the variation will just be in the format. That's an easy case. Or maybe you're asking for search terms, and the variation will be in what the software **does** with the

data. That's harder. Can the software disambiguate one case from another? How?

Each case is unique. Just make sure that the software's response to various input parameters matches what users expect it to do. Test, test, and test again with real users.

Remembered Choices

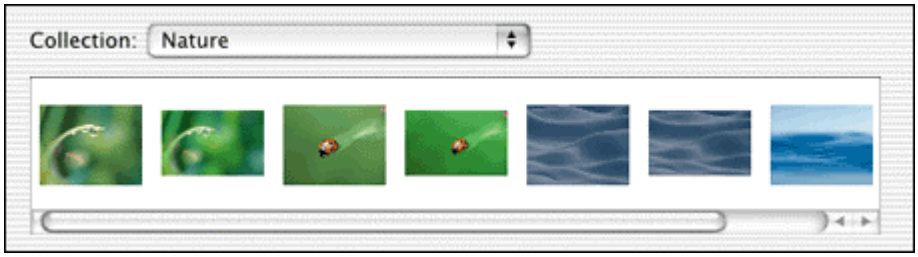
Use when: The user is likely to **reenter text** previously typed, or repeat choices previously made. In particular, they might come back later and have to reenter entire sets of choices or settings.

Why: Save the user the trouble of redoing all these things, and **remember the previous settings** for them. Computers are good at that. People aren't.

How: There are several techniques you can use to do this. From the simplest to the most complex:

- Use the previous value of the control as the **default value.** This works for all kinds of controls -- text fields, dropdowns, radio buttons, lists, etc.
- If you're using a text field, turn it into a **combo box** (which is a combination of a typable text field and a dropdown). Each time the user enters a unique value into the text field, make a new dropdown item for it. If you really want to get fancy, use the dropdown items to do automatic completion on what the user's typing!
- Let the user **save a whole page of settings.** The user names the group of settings and saves it under that name; later, they can choose to load those settings by name.

Illustrated Choices



From the Mac OS X system preferences

Use when: The UI presents a set of choices that **differ visually**, e.g. colors, font families, or alignment.

Why: Why translate a visual concept into words, when showing it visually is so much more direct? You reduce the **cognitive load** on the user -- they don't have to think about what "goldenrod" or "Garamond" might look like -- while simultaneously making the interface **more attractive**. (Hopefully.)

How: Each thumbnail (or color swatch or whatever) should be **accurate,** first and foremost. What the user sees on the illustrated choice should be what they get. Beyond that, show the **differences that matter,** and little else; there's no

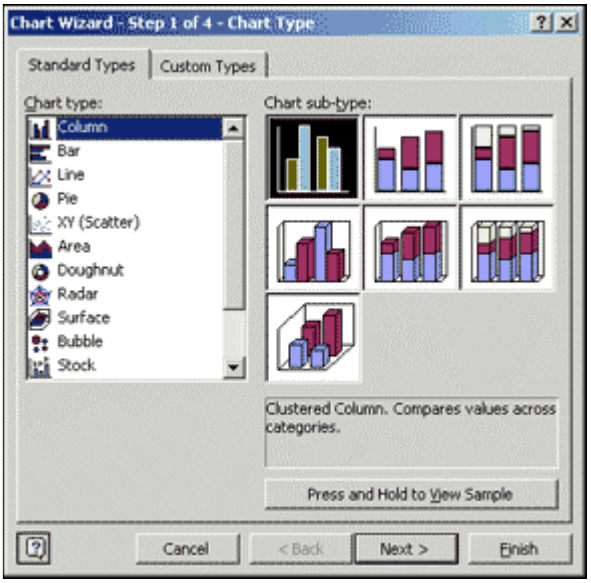
need for perfect miniature reproductions of the choices' effects. Show a simplified, streamlined, and exaggerated picture.

These can be used in dropdown lists, radio boxes, scrolled lists, tables, trees, and specialized dialogs like color choosers. Ideally, you can show the user a set of illustrated choices **all at once,** in one single dropdown or list or toolbar. A user can then compare them to each other immediately and easily. If you show them one at a time -- which sometimes must be the case, as with the *Preview* pattern -- the user sees them sequentially over time, which isn't as good for comparing one to another.

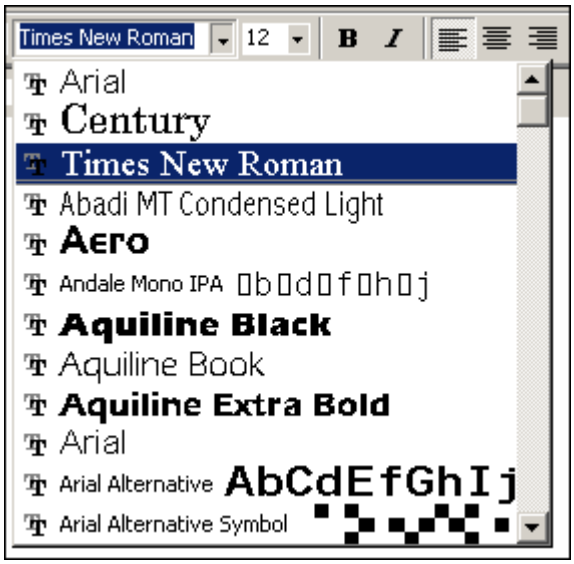
Sometimes it's appropriate to show both the picture and the item's name. If the user would benefit from seeing both of them, do it -- they'll learn to associate the name with the picture, and thus the concept. UIs can teach.

If you need custom icons or thumbnail pictures, consider getting a good graphic designer to do the artistic work. Make sure they are sensitive to the visual vocabulary of the whole application, and that they understand what the choices mean.

Examples:



From Excel for Windows



From Word for Windows

Tables

Sortable Table

Name	Size	Type	Modified
demo		File Folder	8/12/2001 8:38 PM
doc		File Folder	8/12/2001 8:38 PM
frameworks		File Folder	8/12/2001 8:38 PM
javadoc		File Folder	8/12/2001 8:38 PM
lib		File Folder	8/12/2001 8:38 PM
index.html	1 KB	HTML Document	5/1/2001 12:03 PM
license.html	14 KB	HTML Document	5/1/2001 12:04 PM
release_notes.html	1 KB	HTML Document	5/1/2001 12:03 PM
rn_connect.html	1 KB	HTML Document	5/1/2001 12:03 PM
rn_dev.html	25 KB	HTML Document	5/2/2001 4:53 PM
rn_sync.html	1 KB	HTML Document	5/1/2001 12:03 PM

From Windows Explorer

Use when: The UI displays multivariate information that the user may want to explore, reorder, customize, search for a single item, or simply understand on the basis of those different variables.

Why: Giving the user the ability to change the sorting order of a table has powerful effects. First, it **facilitates exploration**. A user can now learn things from the data that they may never have been able to see otherwise -- how many of this kind? what proportion of this to that? is there only one of these? what's first or last? etc. Suddenly it becomes easier to **find specific items**, too; a user

need only remember one attribute of the item in question (e.g. its last-edited date).

Furthermore, if the sort order is retained from one invocation of the software to another, this is a way for the user to **customize the UI.** Some want the table sorted first to last; some last to first; some by a variable no one else thinks is interesting. It's good to give a user that kind of control.

Finally, the clickable-header idiom is **familiar** to many users now.

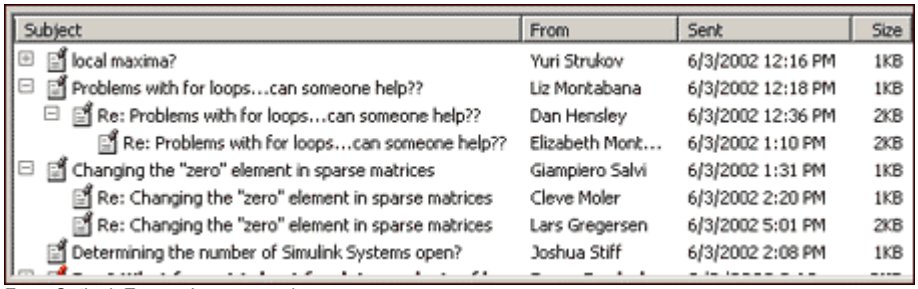
How: **Choose the columns** (i.e., the variables) carefully. What would a user want to sort by or search for? Conversely, what doesn't need to be shown in this table -- what can be hidden until the user asks for more detail about a specific item?

The table headers should have some **visual affordance** that they can be clicked on. Most have beveled, button-like borders. Up-or-down arrows should be used to show whether the sort is in ascending or descending order. (And the presence of an arrow shows which column was last sorted on -- a fortuitous side effect!) Consider using Rollover Effects on the headers to reinforce the impression of clickability.

Try to use a **stable sort** algorithm. What this means is that if a user sorts first by name, then by date, the resulting list will show ordered groups of same-date items that are each sorted by name *within* the group. In other words, the current sort order will be retained in the next sort, to the extent possible. Subtle, but very useful.

If your UI technology permits, the columns may be **reordered** by dragging and dropping. Java Swing has this feature.

Tree-Table



From Outlook Express's news reader

Use when: The UI displays multivariate information, so a table works well to show the data (or allow them to be sorted, as in *Sortable Table*). But the items are primarily organized as a hierarchy, so you also want a tree, or outline, to display them.

Why: Combining the two data-viewing approaches into one gives you the best of both worlds, at the cost of some visual and programming complexity. You can show the hierarchy of items, plus a matrix of additional data or item attributes, in one unified structure.

How: The examples show what you need to do: put the tree (really an outline) in

the first column, and the item attributes in the subsequent columns. The rows -- one item per row -- are usually selectable. Naturally, this can be combined with Sortable Table to produce a more browsable, interactive structure.

This technique seems to have found a home in email clients and news readers, where threads of discussion form treelike structures.

Alternating Row Colors

Song	Time	Artist	Album
Everloving	3:25	Moby	Play
Inside	4:48	Moby	Play
Guitar, Flute & String	2:09	Moby	Play
The Sky Is Broken	4:18	Moby	Play
My Weakness	3:37	Moby	Play
Ginseng Sullivan	4:18	Phish	Live - Unknown date
Cities	5:18	Phish	Slip Stitch And Pass
Wolfman's Brother ->	13:50	Phish	Slip Stitch And Pass
Jesus Just Left Chicago	12:58	Phish	Slip Stitch And Pass

From iTunes for Mac OS X

Use when: A table's rows are **difficult to separate visually**, especially when there are many columns, or multiple lines to a row.

Why: Blocks of gentle color **define and separate** the information contained therein, even when you can't use much whitespace between them. Cartographers and graphic designers have known this for ages. (Remember that colored backgrounds are also effective in defining Titled Sections.) Specifically, alternating row colors help a user:

- **follow a row** from left to right and back again, without getting the rows confused; and
- see the **"footprint"** of the table itself, as separate from its containing page.

How: Pick a pair of **quiet, low-saturation colors** that are similar in value, but not identical. (In other words, one needs to be a wee bit darker than the other.) Good choices are blue and white, beige and white, or two similar shades of gray -- assuming the text on top of them is dark, anyway. Generally, one of the colors is your page's background color.

Alternate the color from row to row. If the rows are thin, you could also experiment with **grouping the rows:** the first three are white, the next three are blue, etc.

This pattern pretty much eliminates the need for horizontal lines between the rows (though you could use that instead, if the lines are very thin and inconspicuous). If your columns are aligned with each other, you don't need vertical lines, either, nor a heavy border around the table -- the row colors will define the edges of the table for you.

Examples:

Rating (5=best)	Title	Submitted	Downloads
3.33 (3 reviews)	**EPS Toolkit for MATLAB** A collection of graphical functions to generate scientific graphics. Stefan Mueller	2000-08-08	6768
4.75 (4 reviews)	**moveplot.m** Enables mouse-based on-screen manipulation of Plot data Brandon Kuczenski	2001-09-17	3618
4.5 (2 reviews)	**Exportfig** Functions for exporting figures Ben Hinkle	2001-09-13	2684
5 (4 reviews)	**pplot** PPLOT is a graphical plot layout and design tool Joachim Johansson	1999-01-18	1552
4 (1 reviews)	**plot crosshairs** A gui interface for reading (x,y) values from line plot(s). Darren Weber	2001-11-08	1145

From http://mathworks.com

Direct Manipulation

Edit-in-Place

Use when: The UI contains **text labels** that the user may want to change sometimes -- the names of objects, for instance. They may be in a graphical editor, or a table, or a tree, or wherever.

Why: If one wants to edit something, it makes sense to try to **edit it where it lives**. Making the user go somewhere else -- a place far away spatially, or disconnected from the original text in another window -- isn't usually ideal; it can be harder to find (though not always), and it takes longer than just clicking on the text and typing in place.

How: When the user clicks or, more typically, **double-clicks** on the text to be edited, simply **replace it with a text field** containing the string (which should immediately be selected). If a text-entry cursor appears in the right place, and/or the text is automatically selected, that may be enough of a cue to the user to start editing; no border is necessary around the text field. Keep it in the same physical location, and retain the display font -- in short, make it **as WYSIWYG as possible**.

One-off Mode

Use when: You're building a **graphical editor**, and there are certain operations -- such as creating objects -- that users **don't normally repeat** or iterate over. Usually,

the user will perform the operation once, then immediately want to do something else, like manipulating the object just created.

Why: Users will **find it annoying** to switch into a mode, do one little thing, then explicitly **switch out** of that mode again -- this often involves clicking on small "hit targets" in palette windows far away from the working canvas. (Here, *mode* is defined as an application-wide state that temporarily changes the behavior of the mouse pointer.) Too much "clickiness" in an interface is a known irritant!

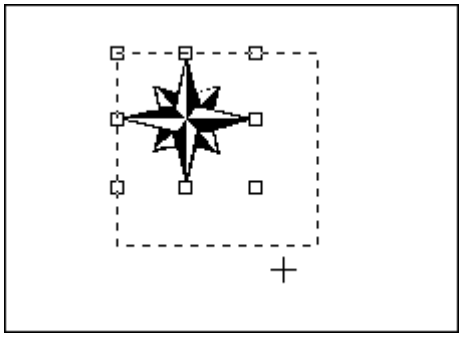
Instead, the interface should do what makes the user's job easier, even if it's not conceptually tidy or easy to program: when the user enters the mode in question, stay in it for **only one operation**, then automatically leave the mode.

How: The hardest part is deciding which operations ought to behave like this. Object creation typically does; zooming, lassoing, paint strokes, etc. typically don't. Find out what graphical editors your users tend to use most, and see what they do.

An example might make all this clearer. Consider a drawing tool in which mouse-clicking on the canvas normally selects objects underneath the mouse pointer:

- User clicks the "Create rectangle" button on a palette. The UI goes into a special creation mode, indicated by a rectangle cursor -- now, clicking on the canvas will mean object placement, not selection.
- User clicks once on the canvas, to place the upper-left corner.
- User clicks again on the canvas, to place the lower-right corner.
- The UI, having counted two clicks and created the rectangle, leaves its rectangle-creation mode and goes back to the default mode, in which clicking means selection.
- User is now free to select objects, move them, resize them, etc. without having to go back to the palette to change mode.

Constrained Resize



From Powerpoint for Windows

Use when: You're building a **graphical editor** which lets the user resize objects interactively. But sometimes a user may want to **preserve the object's aspect**

ratio, for instance -- especially in the case of images or formatted text -- or the position of its geometrical center. Normal drag-the-corner interactive resizing makes this difficult or fiddly.

Why: Quite simply, this can **save the user a lot of work**. If the UI constrains the resize to work in certain ways -- such as by forcing the width and height to remain in the same proportion -- then the user can focus on what they want the built artifact to look like, not on getting the aspect ratio just right.

If the user doesn't have this degree of control over the interface, they might have to resort to doing the resize by typing numbers into a form. That's almost never as nice as doing it via direct manipulation, since it breaks up the user's flow of work.

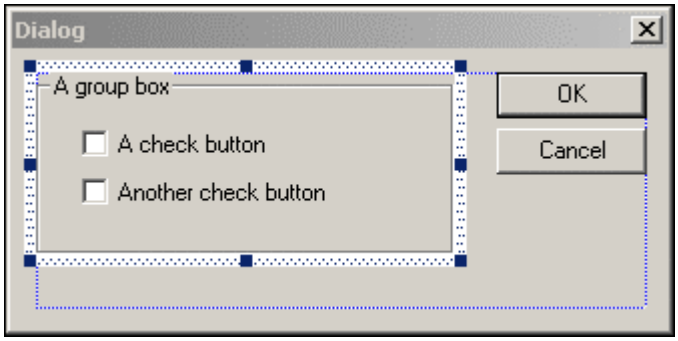
How: This is basically a modified resize mode, so it should behave mostly like your normal resize -- by **dragging a corner** or edge of the object to be resized. Consider using a modifier key to differentiate it from the normal resize, e.g., the user holds down the "Shift" key while dragging. Or if you think most users are always going to want the constraint, make it the default resize operation. Word does this with images.

As the user drags the mouse cursor, a **resize box** should be drawn to show the new dimensions. Whatever kind of constraint you're imposing, show it via the box.

There are many reasons why one might constrain a resize. Some variations on a theme:

- Leave the object's geometric center in the same place, and **resize symmetrically** around it.
- Resize by units **larger than pixels.** For instance, a GUI builder may require that a list box's height be some integral number of text lines. (Probably not a good idea in general, but if that's what you've got to work with, the UI should reflect that.) Or you may be working with a snap-to-grid. In any case, the resize box should "jump" from one size to the next.
- An object may have a **size limit.** Once the user has hit the size limit in one dimension or the other, don't let the resize box expand (or contract, as the case may be) any further in that dimension.

Composite Selection



From Visual C++

Use when: You're building a graphical editor which may contain **composite objects** -- things that can be moved and otherwise manipulated, but happen to contain other objects. This is especially common in GUI builders.

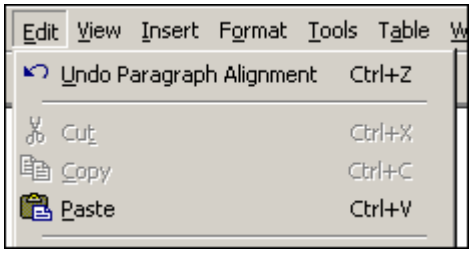
You want the user to be able to "lasso" child objects and create new ones inside the composite, but that means clicking on the composite's background. Should that select the composite, or not? The mouse click has **two interpretations**, reflecting the double role -- parent and child -- that the composite is playing. What to do?

Why: Obviously one of these interpretations has to win out; the user needs to be able to predict what's going to happen when they click the mouse on the composite's background! **Two different kinds of selection** -- one for composites, and one for "leaf" objects that are not composites -- solves the problem (albeit in a rather brute-force fashion). The two selection modes are similar, but respond differently to mouse events like clicking and dragging.

How: Visual C++ seems to have the most elegant solution to this problem. Its group boxes (which, in fact, are not really composites, but that's not relevant to the discussion) can't be selected unless the user clicks near the edge of the object. Mouse clicks inside the object operate on the contents, either by starting a lasso or by selecting a contained object. Dragging the composite is also done via the edges; resizing can only be done via the eight selection handles. This puts some limits on direct manipulation, but it's a simple mechanism, and easily understood once you know what's going on.

Miscellaneous

Smart Menu Items



From Word for Windows

Use when: Your UI has menu items that operate on **specific objects**, like "Close", or that behave slightly differently in different contexts, like "Undo."

Why: Menu items that always say exactly what they're going to do make the UI **self-explanatory**. The user doesn't have to think about what object is going to be affected -- they're also less likely to accidentally do something they didn't mean, like deleting "Chapter 8" instead of "Footnote 3."

How: Every time the user changes the selected object (or current document, or last undoable operation, etc.), change the menu items that operate on it to **include the specifics** of the action. Obviously, if there is no selected object at all, you'll want to disable the menu item, thus reinforcing the connection between the item and its object.

Incidentally, this could also work for button labels, or links, or anything else that is a "verb" in the context of the UI.

What if there are multiple selected objects? There's not a whole lot of guidance out there -- this pattern is mostly seen with documents and undo operations -- but you could write in a plural, like "Delete Selected Objects."

Rollover Effects

Use when: You've put **clickable things** on your UI, but you don't want to make them all look like big clunky buttons. Static visual affordances like beveled borders aren't what you want -- perhaps because they take up too much space, or because they don't work with the design, or or because hunting for clickable items is part of the fun for the user.

Why: Anyone who's used the Web now knows that **if something changes** when you roll the mouse pointer over it, **it's clickable.** The motion of the pointer causes the affordance to appear.

Another advantage of rollover effects is that the user doesn't need to think about whether or not the pointer is really over an object. They gesture at it, and when they see the object or mouse pointer change out of the corner of their eye, they can click on it. Thus it frees up a little more of the user's vision

and attention.

How: There are many, many ways to do this. Naturally, they can be combined (and frequently are). Multiple affordances are better than just one.

- The object's **color** changes, either the text or the background. Most Web sites do this; HTML links do so automatically. Internet Explorer on Windows makes rolled-over buttons go from grayscale to color.
- **Something nearby** changes. Perhaps a title or a short description of the item is shown in a designated place; perhaps an image changes.
- The **mouse pointer** itself changes. UI toolkits provide plenty of built-in cursor shapes for specific situations -- I-beams for text entry, hands for dragging or button-clicking, etc. Use them where you can. They're standard and expected.
- The object's size changes, or it becomes animated. Please resist the urge. Your users will thank you.

A word of warning: **Don't depend too much** on rollover effects. The clickable things should bear *some* visual resemblance to familiar conventions, e.g. text links down the left side of a page. A user shouldn't be expected to go a-hunting for links that aren't recognizable without moving the mouse pointer across every square inch of the page! If in doubt, test with real users.

Short Description

Use when: There are elements in the UI whose purpose or meaning **aren't obvious** or might be hard to remember.

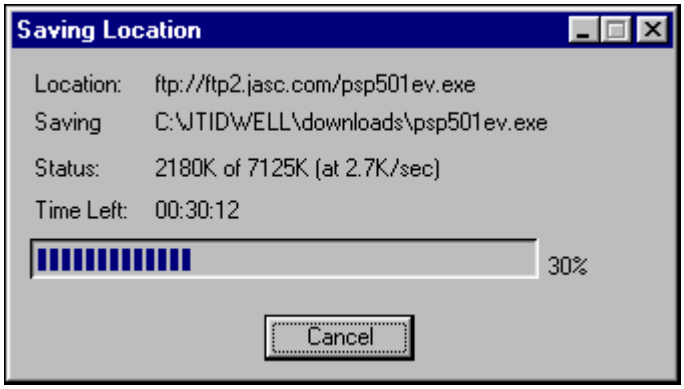
Why: A concise description of the element, placed beside it, makes the UI **self-explanatory.** If the user's never seen this part of the UI before, they have something to go on; if they've forgotten what the element is, the description is a convenient reminder. Thus it also **reduces memory demands** on the user.

How: Write a **descriptive phrase** -- no longer than two sentences -- that succinctly defines the element in the context of the UI. When the user selects the element or rolls over it (see *Rollover Effects*), show the short description, either on the page or in a floating "tooltip."

Normally you write several of these at once, for items in a list or buttons on a toolbar. Use parallel grammatical constructions in the descriptions. For instance, make them all start with a verb ("Cut selection," "Copy selection," etc.) or a noun ("Fill color," "Border color").

In older UIs, you might see short descriptions show up in status bars, at the bottom edges of application windows. These are **too far away** from the item in question -- users rarely saw them! Spatial proximity is important.

Progress Indicator



From Internet Explorer 5

Use when: A time-consuming operation interrupts the UI for longer than two seconds or so.

Why: Users get impatient when the UI just sits there. Even if you're changing the mouse pointer to a clock or hourglass (which you should in any case), you don't want to make a user wait for some unspecified length of time.

It's been shown experimentally that if users see an indication that *something* is going on, they're much **more patient,** even if they have to wait longer. Maybe it's because they know that "the system is thinking," and it isn't just hung or waiting for them to do something.

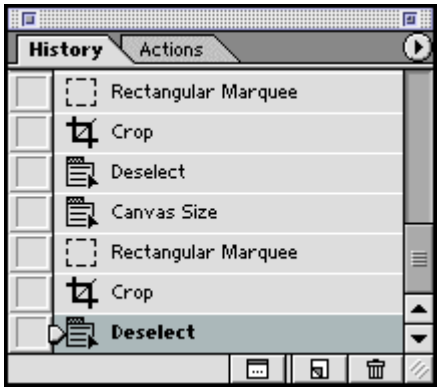
How: Show an **animated indicator** of how much progress has been made. Either verbally or graphically (or both), tell the user:

- what's currently going on,
- what proportion of the operation is done so far,
- how much time remains, and
- how to stop it.

As far as time estimates are concerned, believe it or not, it's OK to be wrong sometimes, as long as your estimates converge on something accurate quickly. But sometimes the UI can't tell how far along it is. In that case, show something animated anyway which is noncommittal about percentages. Think about the browsers' image loops that keep rolling while a page is loading.

Most GUI toolboxes now provide a widget or dialog that implements this pattern, like Java Swing's JProgressBar. Beware of potentially tricky threading issues around these, however -- the progress indicator **must be updated consistently,** while the operation itself proceeds uninhibited. And if you can, keep the rest of the UI alive too.

Command History



From Photoshop 6

Use when: The user performs a sequence of actions in the UI. This seems to be found most often in graphical editors and programming environments.

Why: Sometimes a user needs to **remember what they did** in the course of working with the UI. Computers are good at keeping track of things like that; people aren't. For instance, the user may want to repeat a sequence of operations on a different object.

Furthermore -- and this is just as important, but for different reasons -- sometimes a user will want to **reverse those actions.** Reversibility gives users the freedom to play and explore. They know that whatever they do, they can go back to a known state, and not have to worry about doing permanent damage to whatever they're working with. A recorded sequence of actions enables them to undo as many "levels" as they wish.

How: The software your UI is built on first needs to have a strong model of what an action is -- what it's called, what object it was associated with, how to reverse it, etc. Then you can build an interface on it.

Multi-level undo (i.e., being able to undo multiple operations) should be the first thing you implement, as it's probably the first thing a user will look for if they suspect the UI has a record of actions taken. Then, if you think it's worth the interface complexity, show the list of actions somewhere in the UI. Name them well, and let the user use that list to go backwards some arbitrary number of steps.

The "back" button on a Web browser serves a very similar function as command history.