How to Design Software Good

Haiku is an operating system which is known for its speed and being easy for anyone to use. This is partly because good programmers try to design their apps for more than just themselves. We are going to examine how you can also make your program more appealing. The reason is easy: easier to use means more people using your program. Writing good software can be hard, but it is worth the time and effort.

Who's Gonna Use It?

You probably already know what kind of program you are going to write. If not, put this book away and do some thinking first. Without a clear idea of what you want, it's hard to do something about it. Once you know what general kind of program you would like to create, you also need to figure out who the program is meant for. This can be something as general as 'desktop users' to something as specific as 'Haiku Web Developers'. When you know who the main users of your program will be, you can make certain assumptions about what your users know. You can't necessarily expect a musician to understand how to effectively use a 3D modelling program as advanced as, say, 3D Studio Max, but you can expect them to have skills which lend themselves to using a program for writing music.

Depending on how concerned you are about details, you may even want to create a user profile -- a fictional idea of an example user. This can consist of just one or two sentences or can be several paragraphs. A short user profile contains the person's first name, occupation, level of expertise, and what kinds of things they want to be able to do with their computer. One thing to be sure of is to make the user profile believable -- like a person you might know. In fact, when you design your app, it may be helpful if you know someone who fits into the target audience. You don't want to design your app for that person specifically, but, rather, someone just like them.

What will the User be Doing?

Most people will use your software to get work done, unless, of course, you're writing the next hit game, and despite experiences you may have had which might make you wonder, users are fairly intelligent. The saying that no one reads manuals is mostly true: people generally have better things to do than read software manuals except when either in trouble or as bathroom reading. Try to keep in mind how busy most people are and how valuable their time is to them when you are designing your program.

In order to make doing work with your program easier, you need to prioritize what kinds of work they will do in order of how often the work is done. Come up with a list of tasks the user will be doing and put them into one of 3 categories: Common, Uncommon, and Rare. Common tasks, unsurprisingly, are those things which the user does all the time. In a word processor, this would include typing in text, changing font sizes and styles, and adjusting margins. Uncommon tasks are those which the user does just from time to time. Depending on the user, this might be something like printing labels, adding color to a table, or counting the words in the document. Rare tasks are those which are done every once in a great while. Often times, these tasks are ones which the user would quite likely not miss if it were not possible to do it with your program. This would also be the kinds of things that only 20 percent of your users or less would end up using. If you do decide to include features to handle Rare tasks, think carefully about each task's importance in the grand scheme of things before deciding with any finality.

To paraphrase Einstein, you must make your software as simple as possible but no simpler. Complexity must be kept to a minimum. One major way this can be done is by avoiding features which are used only in remote instances -- the tasks in the Rare group. Design your application to meet the needs of 80 percent of your target audience. Features add complexity and complexity adds difficulty.

How will Your Users Do Their Work?

Once you know who your users are and what they will be doing, you will need to figure out how it will be done in a general sense. This does not include what your program will actually look like, however. In a personal finance application, one of the user's primary tasks will be entering in transactions. The user will need to type in the data for each transaction or download them from their bank over the Internet. You need to know this kind of information for each task the user will have to perform, common or otherwise.

The actual implementation is determined last, which is why it has not been discussed until now. This is where the general usage of the application is determined. For example, the user will have a form to enter transactions into an account. The form should have text fields for each of the different kinds of data, such as the payee, the amount, and the check number. Somewhere nearby, the user will be able to see a list of all the transactions in the current account. Because this is where the actual building of the look of the program takes place, it is crucial at this point in development to know and understand what makes software truly easy to use, so this is what we will examine next. Note that what technologies will be used to actually create the program have not even been mentioned yet. For your audience, technology is merely the means to an end -- a tool -- and not the goal itself.

Summary

Careful planning is the key to writing excellent software, regardless of what stage of development a project may be in. In order to be easy, good software only has what is really needed. It also helps the user do his work wherever possible. Only by thoroughly understanding what the work is, how it is to be done, and who is doing it can a program provide the best experience possible.

Qualities of Good Software

When it comes to software, the proof is in the usage. If a piece of software requires significant time to learn, the user will only invest the time in learning it if (a) he has no other choice but to use that particular software and (b) the importance of the need filled by the software is much greater than the importance of the user's time. In short, a user will learn only what he must in order to do his job and even then only when forced to do so.

Good Software Focuses on a User's Actual Needs

In Chapter 1, it was mentioned that your program should meet the needs of 80 percent of your target audience. It is impossible to meet the needs of all users in the same software and still retain some semblance of being easy to use. Sometimes a user doesn't even know that he needs or doesn't need a feature and it is up to the designer and developer to figure out what is needed. Most often, it is better to have a small collection of more specialized tools than one which does everything.

Good Software Uses Everyday Language

Jobs in the Information Technology industry are almost always professional positions. One aspect which makes IT a profession is that it has its own body of knowledge and terminology to go with it. Most people have only a rudimentary 'computerese' vocabulary. The thing that they look at when they use a computer is called a monitor or just 'the screen'; the little arrow on the screen is called the cursor, and the thing they type with is called a keyboard. Few users know (or care) what a 'file format' is and fewer still do not know what an 'invalid sector' on a floppy disk might be.

Good programs use language appropriate for the audience. The problem is that quite a lot of software intended for the general desktop user reads like Yiddish for Joe User. In such cases, regular, everyday language should be used as much as is possible. For file management software, 'volumes' are really disks -- even though the term isn't quite accurate, a normal person thinks a disk is a storage container and that volume is what you have up too high at really good parties. Images are 'pictures'. A person doesn't 'kill' a application that has 'hung' -- he 'forces a frozen program to quit'. Details like this may seem minor, but many small improvements in a program's usability can have a profound effect overall. Of course, if your target audience is IT professionals, these kinds of terms are perfectly acceptable. Be sure that you match the everyday language of your audience and when in doubt, err toward using less technical language.

Good Software Does Not Expose Its Implementation

Good software works a certain way because it is the best way to be done or close to it, not because the code was written in a certain way or because it was dictated by an underlying API. An example of this would be if a music composition program has an easily-reached maximum song size because the code monkey who wrote it used a 16-bit variable instead of a 32-bit one. While there are sometimes limitations that cannot be overcome, the actual code written and the architecture used when it was written should have as little effect as possible on what the user sees and works with when using your software.

Good Software Uses Graphical Controls Properly

GUI controls were meant to be used in certain ways, and when a program does not use them in that particular way, it is confusing to the user and possibly hazardous to the user's data. Check boxes, for example, are commonly used in lists where a value can be turned on or off. Some programs, however, uses them for choosing one option in a list even though radio buttons are meant for this task. Text boxes should not be used for labelling other controls. This might seem to be common sense for most, but, rest assured, there are many programs which do not do something as simple as this. In short, use standard controls the way they were intended.

Good Software has a Natural Layout to its Controls

Some tasks have a natural, logical workflow, and good programs are designed in a way that capitalizes on this workflow. When entering an address in the United States, the natural order is Name, Street and Number, City, State, and Zip Code. Following any other order would both frustrate the user and also lead to more mistakes in entering data. This also applies to the Tab navigation order of controls in a window. Generally speaking, this order should either be left-to-right, top-to-bottom or in a counter-clockwise circle, depending on how the controls themselves are placed and the expectations associated with the work to be

done.

Good Software Gives Plenty of Feedback

Imagine for a moment that you have just started converting a movie file to another format. You hit the button marked 'Go' and wait. Then you wait some more. You go to the restroom, brew a new pot of coffee, get the mail from the box, confirm that mullets are still out-of-style, and come back to wait some more. You're sure you clicked the button, but nothing seems to be happening. Did something go wrong? Only after some snooping around with a file manager do you find out that everything is OK. You didn't know what was happening because the program didn't tell you what it was doing.

Good software keep the lines of communication open. Good feedback just means making sure that the user knows what your program is doing at any given time, especially if the program is busy doing something which makes him wait. CD and DVD burning programs tell the user how much is left before they are finished making a CD or DVD. File management programs tell how many files are left to copy or move. Web browsers animate a little icon while they download a web page. Users have a natural tendency to think that if nothing seems to be happening, then the program is probably frozen. Making sure that the user knows your program is hard at work puts his mind at ease.

Good Software Makes Errors Hard

It has been said that nothing can be made foolproof because fools are so ingenious. Even so, make it tough for the user to make a mistake. If, for example, the user needs to enter in some text and certain characters are not allowed, then disable those characters for the text box it needs to be entered in instead of nagging the user with a message box. If resizing a window horizontally should not be done for some reason, don't let the user do it. Does your program require a selection from a list before the user clicks OK? Tell the user that -- nicely, of course -- and then disable the OK button until a selection is made. An even better solution would be to select a good default choice for the user and give him the option to change it. Build constraints into your application which prevent errors. This would be why 3.5" floppy disks have a notch in one side -- it can be inserted into a drive only one way. Constraints are also good for lazy developers because then their software crashes less and they don't need to write as much error-handling code.

Good Applications Handle Errors Gracefully

Even if you make it hard for a user to make a mistake, expect your program to have to deal with errors. It is possible for a program to handle errors in a way that doesn't leave the user wondering what happened. When code is written, errors of all sorts need to be anticipated and handled, such as lack of memory, lack of disk space, permissions errors, corrupted files, and loss of network connectivity. As Murphy's Law states, if something can go wrong in a given situation, it probably will. Hope for the best but prepare for the worst: without bordering on the completely ridiculous, handle every error that is likely to occur. Doing so greatly improves the perception of your software by the outside world. Crashes are unacceptable in all cases. Period. Error messages, for example, need to describe at the user's level of expertise what happened and suggest what the user can do to remedy the situation. In the worst case, the program needs to provide an easy way for the user to send technical information about the problem back to you via e-mail or some other means. In all cases, the user's data is to be preserved.

One way that you can see how well your program handles errors is to deliberately try to break it in every way possible. Feed the entire text of your Aunt May's quiche recipe into a text box all at once. Try to open files it has no business being given. Take a valid document, back it up, open it in DiskProbe, enter as much junk data into it as you like, and then try to open it. Break your Internet connection while it's in the middle of an update. Try to using filenames with really wonky filenames. Be creative and ridiculous and, most of all, have fun breaking things!

Good Software is Forgiving

Computers are excellent tools for people because they are good at many things that people are not. From a perspective which focuses on technology, humans are imprecise, illogical, disorganized, and make mistakes frequently. They are, however, excellent at forming habits and matching patterns, two things computers have a difficult time doing. Make commands undoable whenever possible and when it is not possible, be sure to inform the user that such is the case. Capitalize on a computer's strengths to make up for a person's inherent weaknesses.

Summary

Good software takes real work to produce. It is not for the stereotypical 'lazy programmer' unless he knows where it is acceptable to be lazy. In creating something of value to customers, good software creates a positive image for a company and loyal customers who will do your own advertising for you. The best advertising is done by your customers to their friends, family, and coworkers.

Conventions of Haiku

Just as a person generally doesn't go barging into a stranger's home and start redecorating and otherwise making himself at home merely because the owner does not own a shotgun, your program needs to have good manners in getting along with both the operating system and the other programs the user has installed on the system. Some of these are merely good coding practices meant to make your job easier and others are for ensuring that your program can be more easily maintained. None of them are difficult or much work, so there. Now you have no choice but to follow them. :D

Program Settings: Format and Location

While there are lots of ways to store program settings, the easiest and recommended method is placing them in a BMessage and flattening it to a BFile. By using BMessages as your container, you don't have to concern yourself with writing and debugging other code. The exact location used to store them depends on the number of files you will need. If your software needs only one file and will only ever need one, then it is just simplest to place it in the user's config/settings folder. However, should you need more than one file, please put them in their own folder to minimize the clutter. The folder should either follow the format home/config/settings/your_app_name_here or home/config/settings/your_company_name_here /your_app_name_here.

Maintain Responsiveness

Probably the best-known quality about BeOS and Haiku as operating systems is speed. This largely

comes from the extensive use of multithreading in applications. Being responsive does not necessarily mean that your program should never be busy. It should just respond to input and redraw requests even when it's doing something. A common experience encountered by Linux and Windows users is the "blanking out" of a window or an entire program because it is busy doing something else. This comes from two things: the program has one thread of execution devoted to all its windows and that thread is busy doing some sort of time-consuming work. Instead of falling into this trap, which is unprofessional for you and confusing for the user, spawn another thread to do the actual processing and allow the message-handling thread to continue to do its job. This makes sure the user knows that your application is still running properly and has not frozen.

Avoid Hardcoded File Paths

Whenever your program needs to specify a particular location on the system, use of the find_directory() function to generate it. If and when the day comes that Haiku supports multiple users, your application will make a smooth transition to the new architecture. This will also allow for backward compatibility with older versions of BeOS, such as the change in locations for B_COMMON_FONTS_DIRECTORY being in a different place for Haiku than on R5. find_directory() is supported in both C and C++ environments.

Make Your App's Look Fits in with Others

Certain function calls have been provided in the API to aid in making sure that your software shares the same general look as other applications and allow the user to make customizations to the system at the same time. Unless there is a very good reason for it, get colors for your program with ui_color() and the constants which go with it. Determine the size of your controls dynamically - use the ResizeToPreferred and GetPreferredSize for system controls and calculate the size of your own controls based on font sizes obtained from the system instead of hardcoded values. If you're writing code specifically for Haiku and don't care about compatibility with other BeOS flavors, use the new layout API. All of this will allow better ease-of-use for the user who prefers tiny fonts to increase use of desktop real estate and also for older users who need larger font sizes to accommodate weaker visual acuity. Graphics are an important part of a program's look, but don't reimplement the look of the buttons and other standard controls just to make your application stand out from the rest. By keeping visual consistency with the rest of the operating system, you avoid confusing the user with buttons that do not look like you can click on them, strange-acting menus, and so forth.

Live Updates

One way to make sure that your application communicates effectively with the user is to provide "live" updates to information in it. This mostly relates to files in the system. A good example is an address book program which automatically removes and adds entries when new People files are added to the People folder. The information doesn't even have to be data that is outside your program. It could just be as simple as updating an entry in a list of items as the user types makes changes to it in a form in a different part of the GUI. Responsiveness like this in a program helps the user feel more in control of the work he is doing and prevents possible loss of data.

Translators

Haiku comes with bits of technology that allows lazy programmers to be lazy and still have their programs be good ones. One of them is the Translation Kit. Merely by including the library and making use of even something as simple as the BTranslationUtils class will help to ensure that at least this portion of your code is free of bugs. In general, you will probably find yourself making use of image translators most in order to load image resources that your program will use. Just remember that translators are system add-ons and that unless your program has installed a translator for a particular file format, depend only on the formats installed by default in the system. As an aside, MP3 is a format which requires licensing for decoding and encoding to be legal, so depending on the MP3 format is not a good idea unless your program deals specifically with it.

Formats to prefer:

Image
JPEG or PNG

Sound
WAV or Ogg Vorbis

Movie
Ogg Theora or AVI

Squeezing the Most Out of BFS: Queries and Attributes

The filesystem that Be created was nothing short of amazing in the 1990s. Even with the gradual evolution of other operating systems having progressed since then, it is still one of the most powerful around. Attributes are a powerful tool which allow you to store data about a file without being part of the file. This kind of power is easily put to use with audio files, such as FLAC, Ogg Vorbis, and the ubiquitous MP3. By attaching attributes to audio files, you can search for them with a query. Queries also leverage the filesystem's power. As long as the attribute you are querying for is indexed in the filesystem, there is no place a file with that attribute can hide. To top it all off, they are also a fast way of search for files meeting a certain criteria.

Although both attributes and queries are very powerful, use good sense. It takes quite a while to read a large number of attributes from a file, so you may wish to cache them if you are writing a program which will need to read more than just a few of them. Queries can only make use of attributes which are indexed, and their speed goes down as more and more attributes are indexed.

The System Tray: It's Not Just for Dinner Anymore

Because it can very easily become cluttered, install icons in the Deskbar's shelf only when it provides information that is updated often or if it provides functionality which will be frequently accessed by the user. Examples of these kinds of situations would be monitoring memory and processor load, checking mail, or quick access to features provided by a personal information management program. If it is unlikely that the user will need to access the information or functionality less than once per session, don't use the Deskbar -- accessing your program through the Be menu or an icon on the desktop should be sufficient.

Tracker and its Uses

Tracker can be quite a useful tool for your program which requires little effort to provide some courtesies to the user which would otherwise be a lot of work. The Open and Save windows have already been done for you. Please use them. Note that if just a certain type of entry is needed, such as a folder or specific type of file, make a filter that shows only those types of entries. By removing unneeded items from the file navigation window, you are reducing the number of choices the user must pick from and also preventing him from opening the wrong kinds of files. Keep in mind that there may be files that may not have had their MIME type identified, so do not exclude them, either. You can also make Tracker show a window for a particular folder in order to show the user where a particular file has been stored and give him access to it directly.

Getting Input from the User

While it might seem a simple case to get input from the user, it is often not the case if you wish to account for many external factors, such as accessibility, speed, and user expertise.

Mouse Vocabulary, AKA "What's This Button Do?"

The mouse, while the favorite rodent of most users, is not a very intuitive device: mouse skills must be learned. Most of the time, mouse skills are not an issue because the user is clicking on buttons, menus, and so forth, but sometimes a program must deal directly with mouse clicks and moves. Mouse operations fall into one of three categories.

Skills you can expect the user to have:

- Single-click
- Double-click on primary mouse button
- Single click with secondary mouse button
- Drag with primary mouse button

Skills the user might have:

- Drag with secondary mouse button
- Single-click with tertiary mouse button
- Single-click on primary mouse button with modifier key
- Triple-click with primary mouse button
- Scroll with scroll wheel
- Scroll with scroll wheel and modifier key
- Drag with the tertiary mouse button

Don't Use These:

- Click-and-a-half: Mouse down + mouse up + drag with a second mouse down
- Multiple click with non-primary mouse button
- More than 3 consecutive clicks (quadruple clicking or more)

- Drag with tertiary mouse button
- Drag with modifier keys
- Secondary/Tertiary click with modifier keys

In the event that your program must deal with directly handling mouse input, there are certain rules of thumb for how each of the expected skills is to be used. Single clicks are used for the most common actions in operating a control. This would be selecting items in a list, pushing a button, choosing a menu item, and placing the blinking text cursor in a BTextView. Double clicks are used for secondary functions, such as invoking an item in a BListView or selecting a word in a BTextView. Secondary mouse clicks (most commonly called "right-clicking") are generally used for a context-sensitive menu or, in the case of graphics programs, drawing with the secondary color. Mouse dragging is used for selecting text or moving things around. Mouse dragging with B_TERTIARY_BUTTON is generally used for panning. The scroll wheel by itself should be used just for scrolling, but your program may have a use for the scroll wheel when used with a modifier key, as below:

Scroll Wheel +

B_COMMAND_KEY:
Zoom in/out

B_OPTION_KEY:
Scroll by one full page

B_CONTROL_KEY:
Increase/decrease font size

Keyboard Accelerators: The Sports Cars of GUIs

More advanced users often like using keyboard shortcuts for common tasks because they reduce the considerable amount time spent switching between the keyboard and the mouse.

Standard System Accelerators:

B_COMMAND_KEY + N:
New document

B_COMMAND_KEY + O:
Open document

B_COMMAND_KEY + S:
Save document

B_COMMAND_KEY + Shift + S:
Save as… (show Save dialog window)

B_COMMAND_KEY + P:
Print

B_COMMAND_KEY + W:
Close active window

B_COMMAND_KEY + Q:

Quit

B_COMMAND_KEY + C:
Copy

B_COMMAND_KEY + X:
Cut

B_COMMAND_KEY + V:
Paste

B_COMMAND_KEY + F:
Show find window

B_COMMAND_KEY + G:
Find again

B_COMMAND_KEY + B_OPTION_KEY + F:
Show Replace window

B_COMMAND_KEY + L:
Replace and Find

B_COMMAND_KEY + ,:
Settings window

B_COMMAND_KEY + B:
(Word processor) Bold font

B_COMMAND_KEY + U:
(Word processor) Underline font

B_COMMAND_KEY + I:
(Word processor) Italicized font

B_COMMAND_KEY + A:
Select all

B_COMMAND_KEY + Z:
Undo

B_COMMAND_KEY + B_SHIFT_KEY + Z:
Redo

Tab / Shift + Tab:
Keyboard navigation

B_CONTROL_KEY + Tab:
Switch programs

When choosing keyboard accelerators for your program, use them only for commonly-used tasks. Do not use one of the system combinations for a task different from the list above. Doing so will only confuse and frustrate users. Choose key combinations which are easily associated with the functions that go with them. For example, Tracker uses Alt + Up to go to the parent folder of the current window. The Command key should be the main shortcut key. The Option key is generally used to modify an existing shortcut with a closely related function. Command + F shows the Find window. Command + Option + F shows the Replace window.

Design to Prevent Fitts

According to Mr. Fitts, when using a mouse or other pointing device, the amount of time it takes to point to something is proportional to how far away that something is and how big it is. While it might seem obvious, this is often overlooked when a program is designed. The easiest places for a user to click are the four corners of the screen and the pixel directly under the cursor. The reason for this is because the mouse need not be moved for to click on the pixel under it and the user does not have to think much when moving the cursor to a corner because as soon as it reaches an edge, it can go no farther in that direction regardless of how much the mouse is moved. The Deskbar is in one corner of the screen for this reason.

When you design the graphical interface for your program, make controls easy to click on. Toolbars that have a text label as part of each button are inherently easier to click because the labels add to the size of the buttons. A number of existing image editing programs use the secondary mouse button for a pop-up menu to access common features because the mouse doesn't have to move in order to bring the menu up. Researchers have even experimented with circular menus -- called pie menus -- which capitalize on Fitts' Law in order to make a menu as fast as possible. Your program need not go to such measures, but do keep in mind that teeny controls slow users down.

Dynamic Data Exchange: Pass Data Between Applications

Drag and Drop

Not to be confused with falling reptiles, the ability of a user to click on an object, drag it to some target, and drop it to make the target do something with it is both familiar and empowering, just as are other direct manipulation methods. Nonetheless, the drag-and-drop paradigm should be used only where it makes sense to do so -- rearranging items in a list, file management, opening a file dragged to your app from Tracker, etc.

If you plan on supporting dragging within your program or to other programs -- even just a little bit -- you should support drops to the most probable targets in your program. This shouldn't normally be all that difficult because drops to your program will come from either Tracker (in the form of an entry_ref) or from within your own program. Drops from other programs are not required, but if they are especially convenient, they can be a nice bullet point in a marketing blurb for your program.

When implementing dragging, be sure that your program provides feedback about what is being dragged. This is often best done by using an image which closely represents what is being dragged. If this is not possible, a rectangle which is the size of the dragged item's selection should be used in place of the picture.

A feature which is very helpful to a user is drop feedback. Any time the user is dragging something and the cursor passes over your program's window, it can react in a way to show the user that your program will accept the dragged object. Drop feedback can take a variety of forms:

- A list control could draw a line in between two items to show where the dragged item would be placed if the user released the button.
- BTextView controls show a line where dragged text would be placed if dropped.
- If the user drags an entry over a folder, Tracker shades it slightly.

- A color well could highlight its border

There are other possible ways, but these should be enough for you to get the idea. As mentioned before, feedback is a Good Thing (TM). It allows the user to better understand how to use your program to get something done.

The Clipboard

Another way of getting data from one program into another is via the system clipboard. Unlike other forms of data exchange, using the clipboard requires very little effort or code because you have to do is to plunk data into a particular BMessage. Some of the standard controls already do it for you. The BTextView class already implements copying, cutting, and pasting by way of keyboard accelerators. In cases like these, all that you will want to do is add some menu items for less advanced users.

Standardized Clipboard Data Formats

Plain Text

It is easiest to describe the basics with this code snippet. clipboard->AddData("text/plain", B_MIME_TYPE, your_text_string_here);

Styled Text

Use the same format as for plain text but add a text run array field to the message also. You probably won't need to worry about this unless you're writing a word processor or something similar. clipboard->AddData("application/x-vnd.Be-text_run_array", B_MIME_TYPE, run_array, run_array_size);

Generic File References

If the amount of data you wish to pass to another program is too large to place on the clipboard without straining the system, dump the data to a temporary file and place an entry_ref which points to it on the clipboard instead. Audio and video files commonly suffer from this problem. To do this, call AddRef() with the field name "refs" and add an entry_ref pointing to the file you wish to pass this may. Multiple entry_ref objects can be sent just by doing it more than once. If your program is going to handle these kinds of drops file references, it should look for multiple entries in the 'refs' field unless it would be inappropriate to do so.

Image Data

If your program uses a different kind of data than what has already been standardized, there are some guidelines you can follow so that other programs can use it. First, make sure that the data that is put on the clipboard is in as generalized a form as possible.

Field Type	Name	Descriptions
B_STRING_TYPE	class	"BBitmap"
B_RECT_TYPE	_frame	A BRect containing the bounds of the bitmap data in pixels

B_INT32_TYPE	_cspace	The color_space constant for the image data, such as B_RGBA32
B_INT32_TYPE	_bmflags	The Flags() argument from BBitmap
B_INT32_TYPE	_rowbytes	The number of bytes per row, including padding
B_RAW_TYPE	_data	The raw image data
B_POINT_TYPE	be:location	The location from which the image data was copied. May be ignored.

Replicants

Replicants take their cue from Dolly the sheep in allowing a BView to be cloned and allow you to do Really Neat Things(TM). As cool as replicant technology is, it is only an emerging technology in other operating systems and is unfamiliar to most users in general. As such, it is best left as an extra feature and not the primary mode of operation for a program. Here are some guidelines for using them, however.

0. It is appropriate for program to be a replicant if it is a lightweight program which provides information or a feature which the user will want to be able to access frequently.
0. Allow for the size of the dragger handle when computing layout
0. Be sure it is big enough to not get lost when placed on a busy desktop background. At the same time, do not take over the user's desktop unless he wants this to happen. 32 pixels square is a good minimum size, for example.
0. Provide a reliable way for the user to control your replicant. Do not rely on the menu provided by the dragger handle because the handle itself may have been hidden. Instead, provide another means which is immediately obvious or, at the very least, show a pop-up menu if the user clicks on it (with either button) and there are no other clickable controls.
0. Place a frame around the replicant's border so that it stands out from its surroundings
0. Do not make it too visually distracting. This mainly amounts to avoiding lots of bright colors for the controls and limiting the amount of animation and movement.

Use of Text in the GUI

Almost without exception, if you are writing a program, you will need to pay at least a little attention to how it uses text. There are, believe it or not, right ways and wrong ways, and while most of the guidelines for text might seem trivial, paying attention to what seem like niggling little details is what sets a good program apart from the rest.

Above all else mentioned in this chapter, use language appropriate for your audience -- most of the time, this means avoiding technical terms -- and be both clear and concise.

Error Messages, or, "I'm sorry Dave, I'm afraid I can't do that"

Perhaps the place where you should use text the most is in error messages. They should appear as seldom as possible because you anticipated and handled as many error conditions as possible and then tried to blow it up real good, right? ;) When your program can't handle a particular error, the error message given to the user should do the following:

0. Explain what happened in everyday words.
0. Provide enough information to know what happened without providing details which could confuse the user. For example, if a mail client sends a request to a server for e-mail and the server fails to respond, a way to explain this might be something like "MyMailApp could not check your e-mail. The mail server did not respond when contacted."
0. Offer suggestions to help the user fix the problem, if possible. Using the above example, one possible suggestion might be "Try checking your Internet connection with your web browser. If your web browser works, the mail server might not be working correctly and you may want to try again later."

...Ellipses...

An ellipsis is a series of 3 dots (…) used to tell the user that a control, often a menu item or button, will open a window. For example, a menu item named "New…" will display a window which has the title "New". However, if creating a new document does not require showing a window, then an ellipsis should not be used. Please be sure to use the B_UTF8_ELLIPSIS character instead of 3 periods. Some BeOS keymaps, such as the US QWERTY keymap, allow you to type in an ellipsis with the Option + period keyboard shortcut.

Abbreviations, Acronyms, and Contractions, oh my!

Sometimes space is at a premium. Abbreviations, acronyms, and contractions can come in very handy in these instances, but they can also be confusing. Whenever possible, avoid using them. Many times the reason there is not enough space is it isn't being used as efficiently as possible. When you use an abbreviation, please be sure that it is absolutely necessary, that it is both common and clear, and that it is appropriate for your program's target audience. For example, using the octothorpe (# symbol) is an abbreviation for the word 'number' in the English language which fits these criteria in most cases. These same guidelines also apply to acronyms. For example, the acronym CMYK (Cyan, Magenta, Yellow, Black) is acceptable in a color picker for an image processing application designed for graphics professionals, but not in one meant for children. Menus and button labels should never be abbreviated or contain an acronym.

Special care must be used with contractions. They can be a pitfall for users who are not using a program in their native language. Because they require a more advanced command of a language, avoid using them in a place where their role is crucial in conveying the meaning of a message. For example, a checkbox for a message dialog with a checkbox marked "Don't show this message again" which is unchecked by default should be reworded "Always show this message" and have the checkbox checked by default.

Capitalization and Spelling

Nothing is more unprofessional than spelling and capitalization errors. If spelling is not your strong suit, consult a spell checker, dictionary, or at least a friend. This is particularly important if you are working with a language which is not your native one. Use sentence capitalization in all places, that is, follow the normal grammar rules of the language. For English, this means only to capitalize specific names like "Tracker" or "Deskbar".

Use of Fonts

Good use of fonts can way make your program more visually appealing, but when not used well, can make it ugly, hard to read, or worse. Stick to the plain, bold, and fixed system fonts to maintain some general visual consistency across the operating system. It is perfectly acceptable to use different font sizes, but do it only when you need a heading or something similar and try to limit the number of different sizes to just a few. A window which has lots of different font styles and sizes is harder to read and looks unprofessional. Control labels should always be in the system plain font and size specified by the system.

When calculating the layout for the controls, be sure that you do not depend on the system font being set to a particular font size. Most BControl-derived controls implement the methods GetPreferredSize and ResizeToPreferred to take font size into account. A basic idea of the line height for a font can be calculated by calling BFont::GetHeight() and adding together the values of the font_height structure.

Lastly, do not depend on the user having a particular font installed on the system unless you are willing to ensure that the user has it installed on the system. Font handling for your program is your responsibility, not the user's. If you do follow this route, be sure to check to see if you have redistribution rights for the font's license for the font in question.

Special Case: The Command Line

When designing a program to work in a command-line environment there are some special considerations which must be addressed. Interaction with other programs, use in shell scripts, and a generally consistent interface for command-line programs are just some of the things you will need to keep in mind. The major design decisions you should make are how your program will get its data, what options will be available, and what feedback the user will be given.

Spend some time thinking about how your program should interact with the shell and with other programs. Most of the time, this will boil down to whether your program is file-oriented or stream-oriented. Stream-oriented programs like grep and less work pretty much like filters, getting data from stdin and dumping data to stdout. File-oriented programs like zip and bzip2 are given a list of files which the program then operates on. Your program can certainly do both, but choose a primary method because it will influence design decisions later on. Seriously consider handling wildcards instead of relying on the shell to do it for you if your program is file-oriented. This does mean more work for you as a developer, but it also reduces typing and, thus, typing errors for the user. Of course, if it doesn't make sense to support wildcards, then don't do it.

Choose options which are going to be accessible by command-line switches carefully. Make each one available only if it fills a reasonably common task. From the perspective of the user, adding an option is adding a feature, which will, in turn, increase the complexity of your program. Provide GNU-style (double dash + long name) switches for all options. The most commonly-used options should also have a short (single-dash + single letter) counterpart. The switches --help and -h are reserved for showing help information. Only standard UNIX applications (ls, tar, df, etc.) are not required to follow the standard for -h in order to avoid breaking backward compatibility. All new command-line programs need to follow this. Also, if your program requires one or more parameters, do the user a favor and show the help message if there are no extra parameters instead of telling the user to retype the command with the help switch.

When an option requires a particular value, there is also a standard for how the user is to provide the information. GNU-style command options should follow the format --option=value with the option to enclose the value in quotes. Multiple values for a switch should be comma-separated. Short-style command options should place a space between each value that follows it like this: -t value1 value2 value3 ... As mentioned above, wildcards should be handled by the program except when it does not make sense. If your program does something which modifies data, make sure that your program requires some sort of parameter -- a switch, a file, or whatever -- so that data is not lost if the user invokes your program without knowing what it does. You can assume that the user is sharper than a bowling ball, but do not expect the user to have expert knowledge of the operating system or, for that matter, your program. Programs which merely report information -- ls and df come to mind -- are not required to do this as long as the information displayed gives the user an idea of what the program does.

Be sure to give enough feedback when your program does its thing. Like graphical programs, long tasks should inform the user of progress. This may be something as complex as a full-fledged progress meter drawn with text, a simple series of periods, a listing each file operated on, or something else. *All* programs should provide some sort of feedback; the only time a program may print nothing is if there is a "quiet" option which the user has specified. The feedback your program gives doesn't even have to be excessive; you can just give general details. A "verbose" option should provide more detail than the program does by default. As explained earlier in this chapter, error messages should be no more technical than absolutely necessary and should be helpful whenever possible. If your program requires a parameter of some sort and isn't given any, show either the same message as for the help option or an abbreviated version which is still reasonably informative along with something to point the user to the help option for more detailed information.

Branding - Program Icons, About Windows, Graphics, and Other Visuals at the OK Corral

With all this discussion, you may be beginning to think that having a well-designed program means that it is going to be boring to look at. You can definitely have it more interesting to look at than drying paint. There is plenty of freedom for creativity along with some suggestions to help get you started.

Program Icons

Your program's icon is one easy way to set it apart from the rest of the pack. BeOS-style icons follow one of two perspectives - flat and isometric. Flat icons look like a head-on view. Isometric icons "look down" on the icon from a point above and to the right of the object with angled lines being about 30 degrees from horizontal. A good icon can give your program a favorable and professional impression to people who otherwise doesn't know a thing about you or your program. Take some time to create or find a good-looking icon. Whatever you do, don't just slap together a shabby-looking icon. It would be better not provide an icon at all and rely on the system to show the default application icon than to have one which reflects poorly on your program's reputation. For more details on icon creation, consult the Haiku Icon Guidelines.

Design Tips

0. Remember that this will be small -- don't use too much detail.
0. Use color combinations pleasing to the eye.
0. Tie it into the general color scheme of your program if it has one.

About Windows... Doorways to Creative Expression

Give yourself some credit in your program: make an About window. They don't need to be especially fancy, but they can be if you are so inclined. It should contain the title of your program, the version, you and any other authors or the name of your company, and copyright information. If nothing else, write a few lines of code to show a BAlert with this information and you'll have enough. If there is a lot of information to show, using a marquee effect to automatically scroll the information or at least a read-only text view with a scroll bar. Do not include information about the computer itself, such as the amount of RAM or processor speed; it doesn't belong here. While you certainly may show something short about the licensing of your program like "Distributed under the terms of the GNU Public License," the full text of the license belongs elsewhere. The window should not have a tab and should either have a button marked 'Close' or simply disappear when clicked. It should also respond to the Command + W keyboard shortcut.

Graphics and Other Stuff

Graphics need not be limited to just the About window and the program icon. You can also use background images in BViews, toolbars, buttons, and other places. There are few hard-and-fast rules here, but there are some tips you might find useful:

0. Follow the Dos and Don'ts for toolbars mentioned in Chapter 12
0. Background images should not be too busy and should not reduce overall readability.
0. If your program uses only a few small pictures, you may want to package them in a resource file to prevent them from somehow coming up missing in the installation on a user's machine. Crazy things happen on people's computers.

Cursors

Predefined Cursors and their Uses

As of this writing, there are only two predefined icons available in Haiku: the hand cursor and the text cursor. The text cursor is used whenever text editing is needed and the hand is used everywhere else. There will be more cursors for other uses at some future time which will include resizing and drag and drop.

Making Your Own Cursors

You can very easily make your own cursors for your own purposes, but do it only if a different cursor will dramatically improve how well the user can work with your program. At the moment, cursors can only be 16 pixels square and be black, white, or transparent. Be sure that the hot spot -- the actual location passed to applications when a mouse button is pressed -- is very obvious. Good hot spots are the tip of the hand cursor, the point of an arrow, or the center of a crosshairs. Use the full dimensions available to increase the cursor's visibility at high screen resolutions.

NOTE: There is not nor will there ever be a busy cursor in BeOS-based operating systems. This is a deliberate design decision. If you have a need for a busy cursor, you need to make your program more responsive. Often you can use multiple threads to eliminate the need for a busy cursor.

Animating Cursors

While Haiku does not provide explicit support for them, you can animate cursors to provide a little more graphical appeal. As with any kind of movement in the display, use animation sparingly so that it is not distracting.

Menus, Menu Bars, and Menu Fields

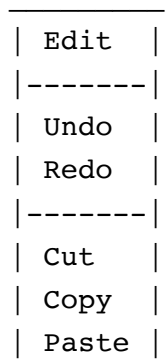
Menus are everywhere, but common as they may be, far too many programs could use them better. Attention to small details in a program's menus is one common difference between good programs and great ones. A developer can pack a lot of features into a small space and still not force the user to remember it all. Menus afford exploration of the interface in steps. This chapter will focus primarily on issues related to menus in general. Pop-up menus and menu fields are discussed in Chapter 12.

Naming and Organization

Choosing good names for menus and menu items is generally not difficult. A menu should have a name which is short and accurately describes the kinds of items it contains. For instance, a File menu should not have Copy and Paste items in it. The name for a menu item should both concisely and accurately describe the function it performs. Items are capitalized as described in Chapter 6 and an ellipsis is used with any item which opens a window. If a menu item opens a window, the names of both the item and the window should be the same.

Two ways to organize menus are the Noun-Verb method and the Verb-Noun method. Noun-Verb names menus after the kind of object that it operates on and items in the menu are actions which can be performed on the object. For example, the File menu contains items such as Open, Print, Save, and Close. Verb-Noun names menus with an action and the items are objects which the action can be performed on. Two example menus could be View and Go. Some "standardized" menus, such as the Edit menu, do not follow either method.

Items should be organized and grouped by function and/or attribute. Use a separator item between each item group. An example of this would be an Edit menu which looks like this:



```
|_______|
```

Undo and Redo perform related, though opposite, functions. Cut, Copy, and Paste are all clipboard functions, so the belong in a group separate from Undo and Redo. A font menu would group font styles together. When organizing the menus in a menu bar, try to have a logical progression from one menu to another. A financial program might have these menus: Program, File, Account, Transaction, and Help.

Submenus are another possibility for grouping menu items, particularly for attributes. They should be avoided when other options exist because they slow the user down and also add complexity to the interface. Younger and older users also have trouble navigating to submenus because of the fine motor skills required. If your submenu has 6 or more items in it, consider placing them in their own top-level menu.

Marking and Toggling Items

Menus can also contain items which indicate the state of a feature, such as the visibility of a tool window. The preferred method is to place a checkmark beside the item indicating the positive state. This is less confusing than to dynamically change menu item labels and has the advantage that it can be used for choosing from more than two states at the expense of screen space. When there are more than two states, all possible states need to be listed in order to prevent confusion.

Examples of Good Toogling/Multiple State Item Usage

```
|Help|
-----------------------
Open manual...
-----------------------
* Show tooltips
-----------------------
Go to the MyApp website
-----------------------

|Font|
-----------------------
Choose font and size...
-----------------------
* Normal
  Bold
  Italics
  Bold italics
  Strikeout
  Underline
-----------------------
```

Common Menus and their Contents

Below are descriptions for menus that are common to many programs. Because they are so common, there are some guidelines to their use so that there is some consistency from program to program. With the exception of the Program menu, each of these menus should be used only if they make sense for your program.

Program: Items related to operating on the program itself.

About <app name here>…

Shows the About window. This is not a commonly-accessed item, so do not provide a keyboard shortcut for it.

Settings…

Show the window which is used to customize settings for your program. This can be a submenu if your program only has a couple of settings.

Quit (Command + Q)

This should be the bottom item in the menu and a separator should go above it. Clicking on this item should close all windows and quit the program.

File: This contains items related to documents handled by your program.

New (Command + N)

Create a new document. This item should have an ellipsis if it shows a window, but not if it doesn't.

Open... (Command + O)

Open a document from disk.

Open recent

This is a submenu in the File menu to allow fast access to recent documents. It should not open a window of any kind except if your program uses a one-window-per-document architecture. The number of recent items should be limited to no more than 5 items.

Close (Command + W)

The function of this item depends on the program architecture. In a program which has one document per window, this closes the window. If there is only one open window, this quits the program. Although it is not recommended, if a program allows for multiple documents to be shown in the same window, this item closes one document.

Save (Command + S)

Save the current document. This should not show a window unless it is a new document that has not yet been saved. It does not normally show a window, so no ellipsis is necessary.

Save as… (Command + Shift + S)

This performs the same basic kind of task as Save, but it always shows a window.

Save all

This item executes a Save command for all documents in the program. The

procedure for handling new documents which have not yet been saved is as follows:

0. Remember the current document window
0. For each document needing a name, bring it to the front, open a Save window, save the document, and proceed to the next document needing a name.
0. When all documents have been processed, bring the window which was originally the active one back to the front.

Revert

Undoes all changes made to the document since the last save.

Import from…

Import data from another file format into a new document. Like Open…, this always shows a window.

Export to…

Convert the data in the current document to another format. Like Save as…, this always shows a window.

Page setup…

Shows the page settings window for printer setup.

Print… (Command + P)

This always shows the print window before printing the current document. This is not intended to be the same as when a toolbar button is pressed.

Edit: Items in this menu are used for different editing tasks

Undo (Command + Z)

Undoes the most recently performed edit. When possible, this item should be dynamic and also include the name of the task that it would undo, such as 'Undo Cut' or 'Undo Typing.' User operations which do not change document data, such as changing zoom levels or the like, should not be included in Undo operations.

Redo (Command + Shift + Z)

Undoes the most recent Undo operation. Like Undo, this menu item should also be dynamic when possible.

Cut (Command + X)

Copies the currently-selected data in the current document to the clipboard and removes it from the document.

Copy (Command + C)

Copies the currently-selected data in the current document to the clipboard.

Paste (Command + V)

Inserts the data on the clipboard into the current document. If there is an existing selection in the current document, the paste operation replaces the selected data with the pasted data.

Select all (Command + A)

Selects all data in the current document.

Search: Tasks in this menu include finding and replacing data and other navigation commands.

Find… (Command + F)

This always shows a Find window for the program. The Find window should then allow the user to choose whatever options he desires for the find and disappear when the actual find is executed.

Find again (Command + G)

This repeats the most recent Find. If no find has been performed in the program yet, it should show the Find window. Because this command does not normally show a window, no ellipsis is needed.

Help: Different ways that the user can learn more about your program and get help when needed.

Open manual…

Shows the manual for the program in a new window. The manual should never be shown in a BAlert.

Go to (MyApp or MyCompany)'s website

Opens the default web browser at the website for the program or the program's company, respectively.

Windows

You Need the Basics

Windows are such common controls that every developer should know the basics of how to use them properly. Some operating systems suffer from the overuse and abuse of windows, such as bizarre error messages, incessant confirmations, wrong window types for tasks, and many other mistakes. Learn the proper ways of working with windows and you will have overcome a major usability hurdle.

Styles and Purpose

Look	Purpose
Document	Windows containing a user document, such as an editor or viewer
Titled	General purpose
Floating	Tool and utility windows

Modal	Modal windows
Bordered	Alert-type dialog windows
Borderless	Splash screens

Feel	Purpose
Normal	General purpose
Modal:Subset	Only when you need to block all windows in a subset
Modal:App	When a user decision is required to continue with the rest of the program
Modal:All	When the user needs to make a system-critical decision, such as system shutdown confirmation. Use this feel only when you absolutely have to.
Floating:Subset	When a subset window needs to take priority in its subset.
Floating:App	Tool and utility windows
Floating:All	System monitors and other windows which the user will always want to have visible. Like Modal:All, use this only when absolutely necessary.

Type	Look + Feel
Titled	Titled + Normal
Document	Document + Normal
Modal	Modal + Modal:App
Floating	Floating + Floating:App
Bordered	Bordered + Normal

Probably 90% of the time you will end up using Titled and Document windows with an occasional floating window. Borderless windows are used frequently for splash windows that are displayed as a program is loading. Bordered windows, which, by the way, do not have a window tab, aren't used much except in BAlerts. Modal windows shouldn't be used more than is absolutely necessary for reasons explained under the Modality section in this chapter.

Naming, Placement, Size, and Other Decisions

Merely knowing what kind of window look and feel to use in a given situation is not enough: you also have to be aware of resizing, zooming, moving, closing, and minimizing because they affect your program in different ways. Once you know what kind of window you need, you should also figure out what its initial size and location are going to be. You also should not restrict the other actions a user can perform on a window unless you have a good reason that does not include "I don't want to write code to handle this."

Care should be given to what name is used for a window. The main window of your program should include your program's name. Windows which were opened from a menu item should have the same name as that of the menu item without the ellipsis. Document windows should include the document's name. The first new document in the application should use the name "Untitled" with subsequent new documents appending a number. A titled window should never have an empty title bar.

The size of a window depends on a number of factors. An application window should have an initial size which is the minimum needed to see all controls in it without overcrowding. Controls should never overlap. This initial size should also be the minimum size for the window which is passed to SetSizeLimits. The initial size for a document window should be large enough to see the entire document or at least a significant portion of the document if it is larger than the screen. Do not arbitrarily restrict resizing unless it

does not make sense to allow resizing in a particular direction. If a window allows resizing, its size should generally be saved when closed or the program quits and restored to that size when shown again.

Zooming is similar to resizing, but there are differences in which should permit zooming and how it should be done. Utility windows, for example, are not intended to be the main focus of the program, so they should not allow zooming even though they should allow resizing except where inappropriate. Document windows should expand to fill the largest sensible space to allow editing. Often times this is the entire screen, but some for some programs, this doesn't make sense. Word processors, for example, will probably move the document window to the top of the screen, resize the width to the maximum width of the current document's view, and resize the height to the bottom of the screen.

The placement of a window on the screen also varies depending on its usage. Program windows should initially either show themselves in the center of the screen or just a little above it. The same goes for the initial placement of a document window. Additional document windows should duplicate the most recent document window's frame offset by 15 pixels in both directions. As with size, a program should generally remember the screen placement of the windows and movement should not be restricted unless there is a good reason for it.

A window need not always be visible. Duh. Considering that it can be hidden or closed altogether may be a bit confusing. Generally, a document should allow minimizing to make it possible for the user to get the document "out of the way" for a moment without having to close it entirely. Utility windows and program windows which are not the main window should normally not permit minimizing and just allow closing. The main window should allow minimizing. Note that when a program is not the focus, utility windows with the Floating:Subset and Floating:App behaviors will be hidden. Windows should normally be allowed to close unless there is a very good reason for it.

A number of times in this document it has been mentioned that you should not prevent something unless there is sufficient reason to do so. Particularly on other platforms, developers have been known to just disallow resizing because they didn't want to bother themselves with writing handler code. There are other instances where, for example, a window could not be closed because the code had a design flaw which would cause a crash if the window were closed. Remember: the more work you do in making your program helpful means the less work the user has to do, which means that your program is easier to use.

B_ACCEPTS_FIRST_CLICK

In MacOS X, the feature this flag enables is called click-through. It means that clicking on the window passes the click through to the control the mouse was under at the time even when the window does not have the focus. As a rule, this behavior should not be used, but there are times when it is quite useful, such as for system monitor programs, the Deskbar, and so forth. Typically, this flag is used for "helper" applications and utility windows.

Modality

By default, windows in BeOS operating systems are modeless, meaning that you can click on other windows besides the one with the focus. Occasionally, there is a need to force the user to make a decision before moving on. This is an appropriate time to use a modal window. The need is far less often than what many would believe, however. Most of the time the only reason to use a modal window would be if not

doing so would make potential for the user to lose data and there is nothing that can be done to the code architecture to prevent it. This follows the "no restrictions for no good reason" way of thinking mentioned above -- modal windows place restrictions on what windows he can or can't click on.

Special Purpose Windows

Alert Windows

Alert windows are the single easiest way to make usability mistakes. Most often, they are used to show error messages. The problem is that they are used *far* more often than they should be and the error messages that are used in them are too technical for a regular user to understand or are just plain useless. They are also used to ask the user a question. More often than not, the question that is asked could have just as easily been answered by the developer, had he thought ahead a bit. Do just about whatever it takes to gracefully handle errors without bothering the user. Only when all other options have been exhausted should you show an error message. For help on writing good error messages, see Chapter 6.

Before you ask the user a question, be sure that it is one which you honestly can't answer yourself or can't do without possibly disturbing the way the user works. For example, it is good courtesy to ask the user if he would like your program to be the default handler for a particular kind of file when your program is installed. It is *never* acceptable to ask the user "Are you sure you want to quit?" If your program is asked to quit, handle unsaved documents and quit. "Are you sure..." questions should be asked only if the action involves the undoable destruction of data, such as deleting a file instead of moving it to the Trash. Avoiding these kinds of situations entirely is a better solution. If the confirmation is asked too often, the user will develop the habit of confirming it without a second thought and your asking the question will be rendered moot.

When you do use an alert window, please follow these guidelines:

0. On a notification, such as an error message, use "OK" for the label of the button

0. When asking a question, make the least destructive of the most common choices the default.

0. Avoid Yes / No button labels. It is much better to use the name of the action in the label, such as Save Changes / Discard Changes. Only in *very* rare cases are Yes / No labels the best choice.

Find Windows

Windows used to search for data in a document are not very different from others, but how they are used can help, hinder, or annoy the user when searching. It only needs to be shown when the user needs to set the search terms. Once they are set, the window should itself disappear and the search should be performed. A find window which stays visible clutters the screen and often obscures part of the current document, even the result of the search. It also places the program in a separate mode, which should generally be avoided. The search itself should default to case-insensitive searches with an option to allow case sensitivity. Search by regular expression is an option that should normally be limited to programs which have programmers as the intended audience.

Settings Window Design

Windows to allow the user to change various program settings are another place where developers commonly commit a usability faux pas or two. The most common mistakes are poorly-chosen defaults, inappropriate language for the target audience, and too many choices. Here are some guidelines for making a good one:

- Choose defaults which fit the most number of people.
- When possible, have changes take place immediately instead of requiring the user to click OK. If you do this, provide buttons to revert changes and also to set the default values.
- Provide settings for significant features. This would include things like default file format for CD ripper, European vs American date format, or the default account in a mail client. Unncessary settings include "Use Ins key for paste" and "Confirm Program Quit".
- Use language appropriate for the target audience as mentioned in Chapter 2. For example, the web browser option "Move system caret with focus/selection changes" requires some technical knowledge. A better way to label such an option would be "Allow text to be selected with the keyboard." Both refer to the same option. The difference is how many people can understand what it does.

Open and Save Panels

BFilePanel is used for both opening and saving files, but there is more to using them well than merely showing a list of files. Remember the location the user was viewing, and if it is inaccessible for some reason, show the Home folder. When possible, make use of file filters to eliminate files your program does not handle. By filtering out "bad" files, you eliminate errors and at the same time reduce the amount of time the user needs to find the file he wants by reducing the number of options. Offering a possible file name when saving a new document is another way to help the user. The title of the panel needs to match the task, whether it is Import, Export as, Save as, Save, Open, or something else.

Controls

BeOS operating systems, while perhaps not as fully-featured as others, possess controls which are the bread and butter of working with GUI programs. Their basic usage is obvious to most, but for the sake of those not sure and for those who want their programs to be the very best, they are covered in detail from the basics to advanced details.

Buttons

Buttons are all around us in the computer world and in the real one, too. A button is used to invoke a command or, much less often, show a window.

Do's and Don'ts of Buttons

Do	Don't
• Use GetPreferredSize and	• Make them too small or ridiculously

Do	Don't
ResizeToPreferred to reduce your work • Avoid using one button for opposite functions without appropriate feedback • Leave sufficient padding around them	large -- like 100 pixels square for the word 'OK' when the font size is 10 point. • Leave them blank • Show a menu with one • Use them for labels • Change the label font

Checkboxes

Checkboxes are like a light switch with a label attached to it. Aside from turning an option on or off, they can also be used for quickly choosing multiple selections in a list of choices.

Do's and Don'ts of Checkboxes

Do	Don't
• Label checkboxes so that the user understands ahead of time what clicking on it will do. • Use care in calculating bounding boxes • Carefully choose label text	• Use them for choosing one item from a list. Use a group of radio buttons instead. • Use a checkbox in place of a button • Use a list of checkboxes as a progress indicator

Radio Buttons

Radio buttons are used to choose an exclusive choice from a list of choices. They can also be used to turn an option on or off if you have plenty of space.

Do's and Don'ts of Radio Buttons

Do	Don't
• Label radio buttons so that the user understands ahead of time what clicking on one will do. • Use care in calculating bounding boxes • Set a default value • Carefully choose label text	• Use them individually or use them the same way as checkboxes • Use more than 5 or 6 in a group

Control Groups

Control groups, namely, the BBox class, are often abused because they are not

well-understood. Control groups are for visually associating different controls which work together for a common function. Unfortunately, they are most often used just for fluff.

Do's and Don'ts of Control Groups

Do	Don't
• Label control group • Use control groups when there are lots of controls in window	• Use a control group to label a single control • Nest control groups • Use a control group around all controls in a window • Mix control group border styles

Sliders

Sliders are a good way to allow a user to choose a value that must be within a certain range, especially if the effect isn't exactly concrete or the value has no meaning to the user. Good uses for sliders include setting volume, mouse acceleration, and movie playback position.

Do's and Don'ts of Sliders

Do	Don't
• Label the ends of the slider • Show tick marks for each value if your slider has distinct increments • Place the largest value at the right or top • Use the appropriate orientation • Label each slider position when the increment size changes, such as logarithmically	• Use a slider for a progress indicator or scrollbar • Label each slider position for sliders with fixed increment sizes

Labels

Labels help the user know what a particular control is for. Most of the controls in the BeOS API include and handle their own labels. This section deals with both the labels included with controls and ones created on their own.

Do's and Don'ts of Labels

Do	Don't
• Label controls	• Use a label for large amounts of static text

Do	Don't
• Use a separate label when layout makes it hard to use a control's built-in label. • Use proximity to associate labels with their controls	• Use a separate label object for controls which have label functions built-in

Text Controls

Text controls are one-line editable text boxes. They are useful for entering text into forms.

Do's and Don'ts of Text Controls

Do	Don't
• Disable characters you don't want in the field • Try hard to validate text entered • Make them large enough to accommodate the length of common entries • Allow clipboard operations when possible	• Use a text control for a label • Make them too small • Use them for static text • Arbitrarily limit the number of characters without a very good reason

Text Views

Text views are multiline text controls. They also provide undo, styled text, and a number of other useful text functions.

Do's and Don'ts of Text Views

Do	Don't
• Disable editing if you use one to display static text • Allow both undo and selection of text unless there is a good reason not to • Allow clipboard operations when possible	• Use one for single-line input • Forget to pad the text rectangle inside the text view

Lists

Lists are used to display lots of individual items, possibly in a hierarchy. They can also allow multiple or single selections.

Do's and Don'ts of List Views

Do	Don't
• Provide a way to select all items in multiple-selection lists. • Sort your list	• Use just enough space to display just a couple of items • Try to use it to select from hundreds of items. Use a search instead. • Use list items to toggle options like a list of checkboxes

Tabs

Tabs can be quite handy for displaying multiple views in a small space. Unfortunately, they seem to suffer from being too easy to abuse.

Do's and Don'ts of Tabs

Do	Don't
• Use tabs to manage and organize otherwise complex dialog windows • Use a list instead of tabs to manage large numbers of views	• Use tabs to select a particular mode • Nest tab views • Use so many tabs that a scrolling is needed to see them all • Place confirm / cancel buttons inside tabs • Overlay toolbars by using tabs • Use vertical text with vertical tabs • Use tabs on more than one side at once • Use one tab all by itself • Use unlabelled icons for tab titles • Use a scrollbar to scroll a form inside a tab

Scrollbars

Scrollbars, unlike tabs, tend to not be abused. Use them to display more items than can fit on the screen, but don't overuse them.

Do's and Don'ts of Scrollbars

Do	Don't
• Adjust range and step size when their target view is resized	• Scroll forms

Do	Don't
	• Try to customize their look • Use them like a slider control • Nest BScrollViews

Menu Fields

Menu fields are buttons which display a label and pop up a menu when clicked.

Do's and Don'ts of Menu Fields

Do	Don't
• Use them in place of radio buttons when space is limited • Remember to follow the menu construction guidelines mentioned in Chapter 9	• Use them instead of a menu bar for the main program menu

Pop-up Menus

Pop-up menus differ from regular menus in that they can be invoked from anywhere on the screen.

Do's and Don'ts of Popup Menus

Do	Don't
• Make pop-up menus sticky • Use pop-up menus for context menus	• Show a pop-up menu by clicking on a button or do other similar nonsensical control daisy-chaining. • Use a pop-up menu as a tooltip • Use the primary mouse button to show a context menu

Progress Meters

BStatusBars are used to show progress in BeOS operating systems.

Do's and Don'ts of Progress Meters

Do	Don't
• Choose a color carefully if you don't use the default • Use them whenever an operation has the potential to take a while	• Use a slider or scrollbar for a progress bar. • Use a progress bar as a control separator

Do	Don't
• Properly label the progress bar	

Toolbars

While BeOS doesn't have official support for toolbars, they are easily constructed because they are little more than a collection of graphical buttons. They are used for providing fast access to commonly-used functions.

Do's and Don'ts of Toolbars

Do	Don't
• Use them to speed up access to common functions • Avoid bright colors • Make the function of each button obvious from the icons used • Make toolbar buttons look like buttons and look clickable	• Add a toolbar just for looks • Scroll toolbars • Use put too many buttons in a toolbar or have too many toolbars on the screen

How to Make a Good Error

By nature of not being perfect, developers make mistakes and no program is without its bugs. As mentioned earlier, conditions which could cause errors should be anticipated and handled as much as reasonably possible. In those instances where there is nothing left to be done except show an error message, be helpful, polite, and as non-technical as possible.

Examples of Good Error Messages

Uh-oh... MyMP3Ripper couldn't create the playlist /boot/home/playlists/foo. This may have happened for a number of different reasons, but most often happens when making playlists on a non-BeOS drive, such as one shared with Windows. Certain characters, such as question marks and slashes cause problems on these disks. You may want to check the names of the artist, album, and songs for such characters and put a good substitute in its place or remove the character entirely.

MyApp can't copy your file to the disk because there isn't enough space for it. If you would like to copy it, move or delete files on the destination disk and try again.

MyApp didn't recognize the data in the file '/boot/home/foo'. It might not be a MyApp file or perhaps the file is corrupted.

(From a website):: O No! If you are reading this message, it means we have made a mistake. Don't worry, you can be sure that as you are reading this people are freaking out around here - testing circuits, rebooting systems, and even typing resumés (just kidding). Please try back in 5 minutes, as we still have plenty of great stuff.

Real World Examples of Bad Error Messages

Wrong Button! This button doesn't work. Solution: Try another.

The project file "e:\ws\foo" may have been modified on disk by the preceding Source Control operation. However, you also have made changes to this project which have not been saved. If you reload the project you will lose your current changes, but if you don't, you risk overwriting the new changes on disk, which is usually much worse. Do you want to reload it now? [Yes] [No]

Mail Engine: No Error

Spell Check Cancelled

Unexpected Error. Please investigate.

System Error 0x80004005 (02147467259). Unspecified error.

Question or information that needs the user's immediate attention.