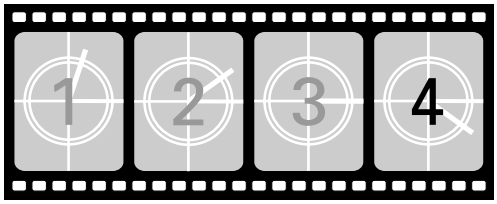


Textual Bloopers

Introduction

Uncommunicative text

Developer-centric text

Misleading text

Introduction

One irony of graphical user interfaces is that most aren't very graphical. The typical GUI contains a lot of text:

- The labels for commands in menus or on buttons are mostly text.
- Instructions are almost always text.
- Most user input consists of typing or selecting words and numbers.
- The labels on most controls and form fields are text.
- The names users assign to data files and other data objects are always textual.
- Error and warning messages are mainly textual, even if highlighted with a color or a symbol.

Therefore, do not underestimate the importance of text in GUIs.

Software designers may try to minimize the use of text in software, but many concepts simply cannot be expressed without text. The saying "A picture is worth a thousand words" is an oversimplification: sometimes a few words are worth more than any number of pictures.

For example, the designers of an interactive movie game wanted the game's navigation controls to be purely graphical, but found it necessary to augment many symbols with text to clarify their meaning [Johnson, 1998].

Even in the most graphical of user interfaces, text usually plays a role. Creative's Surround Mixer application is more graphical than most (Figure 4.1). Nonetheless, it makes use of text: the company logo, the application title, the menus, and the tooltips for controls.

Textual usability problems are usually easy and cheap to correct. On the other hand, they often have root causes in the development process or organization. Correcting those is, of course, not easy or cheap.

Because text plays an important role in user interfaces, there are many ways to use it badly. These are textual bloopers. This chapter describes three categories of textual bloopers, explains why developers sometimes commit them, and provides advice on how to avoid them.

Uncommunicative text

The first four textual bloopers are caused by poor writing. They often result from assigning the writing of text in a GUI to people who are not skilled at that.

Blooper 22: Inconsistent terminology

One of the most common textual bloopers is to be haphazard and inconsistent about which terms are used for what concepts. This makes software much harder to learn.

Many development teams aren't aware that inconsistent terminology is bad, so they make no effort to ensure that their terminology is consistent. What begins as a flaw in their development *process* turns into a flaw in their *products:* many-to-one and one-to-many mappings between terms and concepts.

Eye-opener: List all the terms

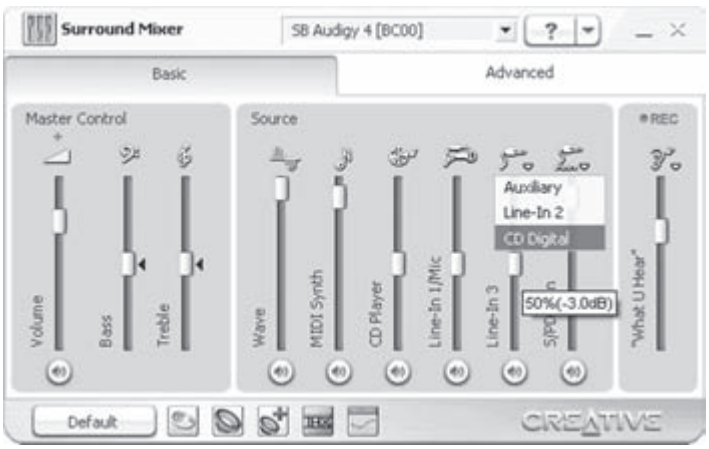
It is useful to construct a table showing the terms used in an application and its manuals for each concept user to see. The results are usually eye-opening to development managers: "I had no idea! No *wonder* users are having trouble learning to use our product!" Unfortunately, the typical reaction from programmers is "So what? We have more important problems to worry about than whether we use the exact same term from one screen to the next."

There are two different ways for terminology to be inconsistent.

Variation A: Different terms for the same concept

The more common variation is for software to use multiple terms for a single concept. It might refer to "results" in one window and "output" in another, even though the same thing is meant. If you were trying to confuse users, this would be one of the best ways.

Figure 4.1



Creative's Surround Mixer: highly graphical, but like most GUIs, includes text.

The following are samples of terms used interchangeably in real applications:

- Properties, attributes, parameters, settings, resources
- Welcome window, Introduction window
- Version, revision
- FAQ (frequently asked questions), QNA (questions and answers)
- Find, search, query, inquiry
- Server, service
- Exit, quit
- Order size, order quantity
- Stock symbol, instrument ID, instrument, instr ID
- Task, step
- Specify goal, define goal

Inconsistent terminology can result from name changes during development that were not corrected everywhere in the software and documentation. It can also result from not creating and using a product lexicon during development. Sometimes inconsistent terms are caused by a lack of communication or a lack of agreement between programmers. Programmers also may not have considered using consistent names important, given the time pressure they were under. All of these causes come from Blooper 67: Anarchic development (page 348).

A very common case of multiple terms for a concept is when error messages use a form field's *internal* name rather than its label on the form. Examples come from CityCarShare.org, Evite.com, and UBS.com (Figure 4.2).

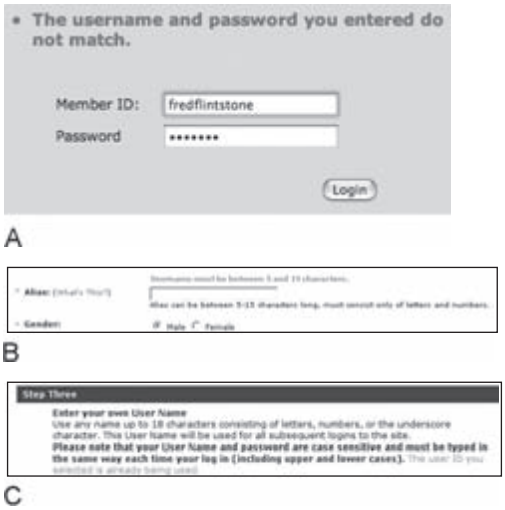
When different words are used to describe the same thing, users may not realize that. Users think mainly about the goal they are trying to achieve and the data they are creating and manipulating (Basic Principle 5, page 35). They think hardly at all about the user interface, such as whether a "User ID" is the same thing as an "Alias." Inconsistent terminology causes users to either make errors or spend mental effort figuring out how terms relate.

Variation B: The same term for different concepts

The opposite error is almost as common: using a single term for more than one concept, also known as *overloading* a word. For example, here are the many things "View" meant in one application:

- The data-display windows, e.g., Understanding View, Evaluation View, Fields Ranking View

Figure 4.2



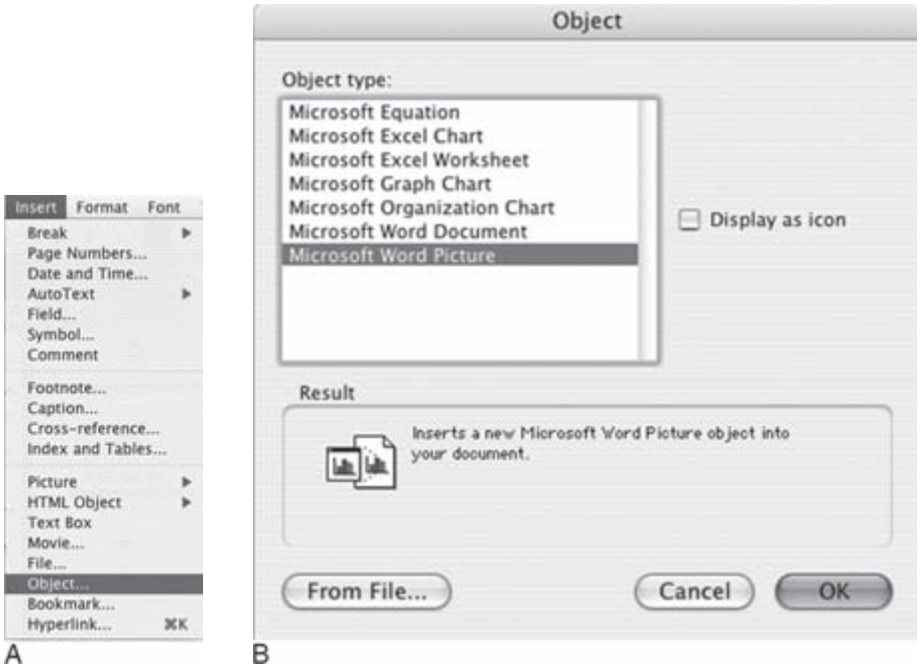
User ID named differently in error messages. (A) CityCarShare.org. (B) Evite.com. (C) UBS.com.

- Different ways of filtering the Understanding View, e.g., Required View, Specific View
- Actions that affect the data-flow diagram, e.g., Shrink View, Enlarge View
- The View menu, which controls display of other parts of the application
- Some items in the View menu treated "View" as a verb (e.g., "View → Results") while others treated it as a noun (e.g., "View → Enlarge").

Using the same term for different things is usually not intentional; it happens because developers don't think about it. In normal conversation, people use words that have multiple meanings, and listeners resolve ambiguity either from the context in which the word is used or by asking the speaker to clarify. Human–computer communication is less forgiving of ambiguity. It doesn't provide much conversational context. Requests for clarification are one-way only: software can ask its user to clarify an input, but the user cannot ask software to clarify the meaning of a word. Therefore, overloaded words are less acceptable in user interfaces than in communication between people.

Microsoft Word has overloaded terms. Word's Insert menu includes both an Insert Object… command and an Insert Picture… command (Figure 4.3). Insert Object inserts many different types of objects, including Equations, Excel worksheets, Word documents, and Word pictures. Users might expect Insert Object with Word Picture as the object type to do the same thing as Insert Picture. Wrong! Inserting a Word Picture using Insert Object inserts a graphics frame for creating line drawings. In contrast, Insert Picture displays a file chooser that lets users import an externally created image file.

Figure 4.3



Microsoft Word: overloaded term "Picture" means both image file and line-drawing area.

Often the same term means both an object and a part of the object. In an e-mail program, the word "message" might sometimes mean the entire file received from another user, including headers, message body, and attachments. Other times, the word "message" might mean just the text content. The ambiguity would confuse new users, slowing their learning.

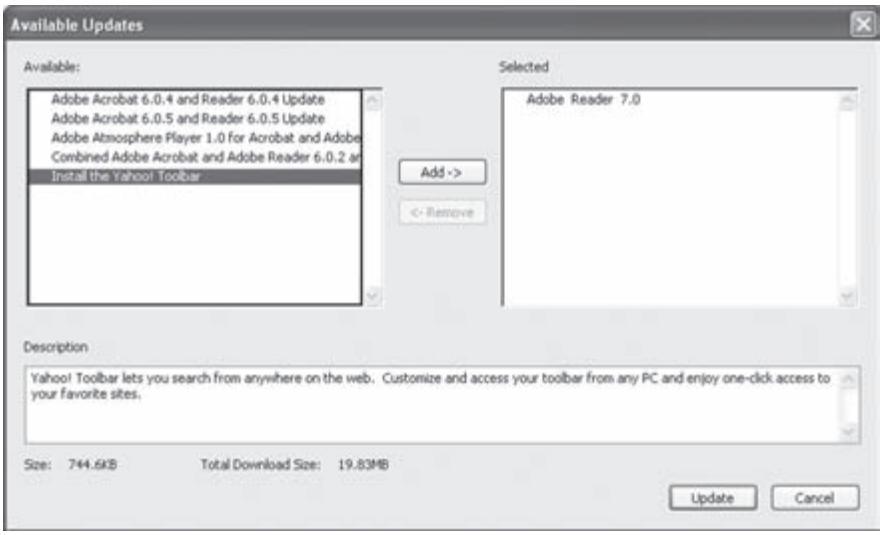
A commonly overloaded term is "select." It often is used to mean both: (a) clicking on an object to highlight it and (b) adding an object to a list. Consider, for example, the Available Updates dialog box from Adobe Reader (Figure 4.4). The list on the right is labeled "Selected." The "Add" button adds an item from the left list to the right list. The item it adds is the highlighted item in the left list. What do we call that item? The *selected* item, of course.

When you assign extra meanings to "select," you open the door to confusing instructions such as:

> To select the updates you want to install, first select them in the Available list and click the "Add" button, which adds them to the Selected list.

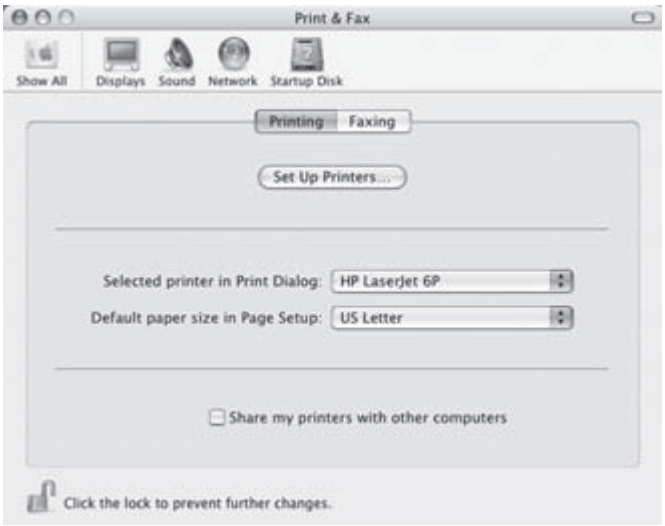
The Print & Fax Preferences window in MacOS X also misuses "select" (Figure 4.5). The first of two settings refers to the "selected" printer, meaning the *default* printer. (We will save for later the blooper of exposing the GUI toolkit term "dialog" to users.)

Figure 4.4



Adobe Reader: uses the reserved term "selected" incorrectly.

Figure 4.5



MacOS X: "selected" misused for *default*.

Confusion between the terms "cursor," "text insertion point," and "screen pointer" is another source of ambiguity. Before GUIs, there was no such thing as a screen pointer, and the text insertion point and the cursor were the same thing. Nowadays these are distinct concepts, but "cursor" sometimes means the text insertion point and sometimes means the screen pointer.

Avoiding Blooper 22

Software users are trying to find their way in an unfamiliar application. They don't know that the programmer of the main window calls searching a database "making a query" while the programmer of most of the dialog boxes calls it "specifying an inquiry." They don't know that an "employee" and a "record" are the same thing or even that "record" is a noun in this program.

Furthermore, computer users don't *want* to know these things. They just want to do their work. They are not interested in the computer and its software per se. They don't care how developers view the software. They are often so focused on their work that if they are looking for a Search function but it's labeled "Query" here, they may miss it. Therefore, design the terminology in a UI as if the users will interpret it extremely literally.

One name per concept

Caroline Jarrett, an authority on GUI and forms design, states this rule for terminology in software and Web sites:

> Same name, same thing; different name, different thing—Caroline Jarrett, www.formsthatwork.com

Terms in software should map strictly 1:1 to concepts. Even terms that are ambiguous in the real world should mean only one thing *in the software*. Otherwise, the software's usability will suffer.

Create a product lexicon

Early in development, you should specify the concepts the software will expose to users. This is called developing a "conceptual model" (Basic Principle 2, page 18). From that your team should develop a product "lexicon."[1] It lists a name and definition for each concept that will be exposed to users in the product and its documentation. It should map terms 1:1 onto concepts. It should not assign different terms to a single concept or a single term to different concepts.

Terms in the lexicon should come from the software's supported *tasks,* not its implementation. Terms should fit well into the users' normal task vocabulary, even if they are new. Typically, UI designers, developers, technical writers, managers, and users all help create the lexicon.

---

1. Also called "nomenclature," "vocabulary," "dictionary," "terminology standard."

Use industry-standard terms for common concepts

Certain concepts in GUIs have industry-standard names. These are the GUI equivalents of "reserved words" in programming languages. If you rename such concepts or assign new meanings to the standard names, you will confuse users.

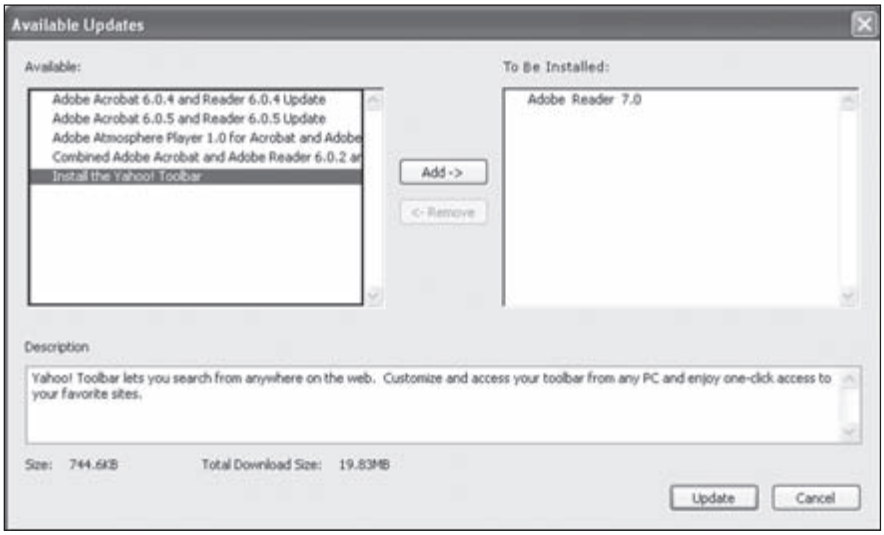
One reserved term is "select." It means clicking on an object to highlight it, marking it as the object for future actions. The word "select" should not be used for any other purpose in a GUI, e.g., adding an item to a list or a collection. Adobe Reader could avoid the blooper by using the label "To Be Installed" instead of misusing the word "select" (Figure 4.6).

Standard terms and their definitions are given in platform style guides, such as the ones for Windows [Microsoft Corp., 2006], Macintosh [Apple Computer, 2006], and Java [Sun Microsystems, 2001]. You should use the industry-standard GUI vocabulary for your target platform.

Enforce the lexicon

The product lexicon should be followed consistently throughout the software, user manuals, and marketing literature. To ensure this, someone has to enforce the lexicon. This can be either the project's information architect or the head technical writer. The enforcer reminds developers to either use the agreed-upon term for a concept or petition to *change the lexicon*.

Figure 4.6



Adobe Reader could avoid misusing "selected" by relabeling the right list "To Be Installed."

"Enforcer" conjures up images of burly men carrying violin cases, but it is better if the enforcer is friendly. Here's one side of a phone conversation between the lexicon enforcer and a programmer:

> "Hey, Anoop, it's Sergei. Got a minute? On your pages in our customer service Web site, you use the term "bug report" for when customers submit a problem. But our agreed-upon term is "action request," remember? That's what's in the lexicon. Where's the lexicon? At the project's Intranet Web site. Can you please change "bug report" to "action request" on all your pages? We're running usability tests on Thursday, so I'm hoping you can make these changes by Wednesday. You will? Great, thanks!"

The lexicon should be treated as a living document: it changes as the product evolves based on new design insights, changes in functionality, usability test results, and market feedback.

Test the lexicon on users

As the lexicon is developed, it should be tested on people who are typical of the software's intended users to see if it matches the users' vocabulary. If it doesn't, change it.

Terminology can be tested before the software is implemented or even fully designed. Users can be shown terms and asked to explain what each term means to them. They can also be asked to match terms with descriptions by arranging 3 × 5 cards or by drawing lines between terms and descriptions printed on paper. Finally, the terminology can be tested in early mock-ups.

Use message files

Before release, a systematic review of all text can uncover both variations of this blooper: different terms for the same concept and the same term for different concepts. However, if the only way to review a program's messages, labels, and instructions is by operating the program or searching through its source code, oversights are likely.

If the text displayed by a program is in a message file,[2] reviewing it and checking it for conflicts and inconsistencies are much easier. That is only one of the many advantages of using message files.

When different parts of the software need to refer to the same concept or present the same message, they should simply reference the same text string in the message file. That reduces the chances of committing Variation A of the blooper: different terms for the same concept.

---

2. Often called a "resource file."

Message files also make it easier to avoid Variation B of the blooper: the same term for different concepts. When the message file is reviewed, duplicate text strings in it are one of two possibilities:

1. *Redundant text strings that should be one:* These are errors. Leaving them separate makes it possible that someone will change one and neglect to change the other, causing the software to manifest Variation A of the blooper. All but one instance of the string should be deleted, and all references to that string should point to that one instance.
2. *Text strings for different situations that are the same:* These are probably errors, giving rise to Variation B of the blooper. They should be reworded so that they differ (while staying true to the product lexicon). A few such duplications might be legitimate homonyms or heteronyms, e.g., a program might use both the verb "refuse," meaning "decline," and the noun "refuse," meaning "garbage." In such a case, consider changing one of the terms, for example, using the word "garbage" instead of the noun "refuse." When translated to other languages, the words that are spelled alike would probably be translated to different words anyway; for example, the verb "refuse" translated to German becomes "ablehnen," whereas the noun "refuse" translates to "Abfall."

Using message files not only enhances textual consistency, it also provides a single place for technical writers to check the text and greatly simplifies translation to other languages.

Blooper 23: Unclear terminology

Even when software uses terms consistently, the terminology can still be unclear and prone to misinterpretation. This can happen in three different ways.

Variation A: Ambiguous terms

Terms that mean only one thing in the software can still be ambiguous. A term may have other meanings *outside* of the software. Users then have to ignore what they know about the term and learn what it means *in the software*.

"Enter" is often used in computer software to mean typing data into the computer. However, "enter" also means "to go into." In computer software and especially in Web sites, that meaning of "enter" may make just as much sense to users as the "type data" meaning. Designers are often so focused on their own intended meaning of a term that they fail to realize that the term has other equally appropriate meanings.

One application had a splash screen with a graphically labeled button that on mouse-over displayed this tooltip text:

> Click here to enter application.

Clicking the button displayed the application's main window. However, novice users could misinterpret the tooltip as meaning that clicking the button would display a text box in which they could enter the name of a software application.

Textual ambiguity can be worse when verbs are used as nouns. A software company developed an application development tool for C++ programmers. The main menubar included the command Build Window. The developers intended this to be a noun phrase: the window for *building* a program—compiling and linking the various modules of the program together—the *Build* Window. Unfortunately, users—hard-core C++ programmers—persisted in reading the command as a verb phrase: Build *Window*. This alternative interpretation—building a window—made at least as much sense in the application development tool as the intended interpretation did. It was a surprise to the designers that anyone would interpret the command that way.

Problems caused by turning verbs into nouns are discussed more fully in Blooper 26 (page 173).

Variation B: Terms for different concepts overlap in meaning

Programmers sometimes use synonyms to name distinct concepts: for example, "delete" for deleting text and "remove" for deleting files. When two different functions have names that are usually synonyms, users have to learn which synonym means which concept.

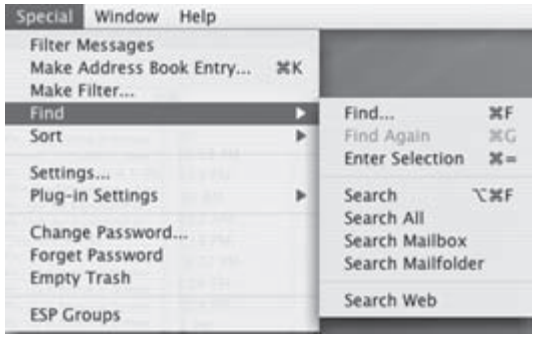
Apple's MacOS uses the word "copy" for copying document content while using "duplicate" for copying document files. Users have to learn this arbitrary distinction.

The e-mail program Eudora (Figure 4.7) provides another example. The "Special" menu (a vague, catch-all name) contains a Find cascading menu with commands for searching stored e-mail messages. The Find menu has the two commands Find and Search, which do different things. Find searches the headers of messages in the currently open e-mail folder and highlights the first matching message. Search searches one or more e-mail folders for messages containing the specified text in any part and lists *all* matching messages.

Here are two more real-world examples:

- An Intranet Web search facility provided two different functions for finding information related to one's previous search. One was Find Related

Figure 4.7



Eudora for Mac: users must learn arbitrary distinction between Find and Search.

Concepts; the other was Find Related Terms. Many users did not understand the difference between these two functions, and some did not even realize that they were different.

- A company developed a Web site for people seeking to buy a home in the United States. Logged-in users could keep two types of notes for future use: (1) preferences for a home, such as price range, size, number of rooms, and (2) an annotated list of homes they were considering. These two types of notes were separate, but had similar names: notes on home-buying goals were in a "Personal Planner," while notes on appealing homes were in a "Personal Journal." Not surprisingly, testing found that users confused these two.

Variation C: Concepts too similar

In Variation B, it is hard to say whether the *terms* were too similar, the *concepts* were too similar, or both. Sometimes, concepts in an application are so similar that users confuse them no matter what they are called. This is not a naming problem, but rather a deeper, conceptual design problem. Therefore, it is discussed fully in Chapter 6, Interaction Bloopers (Blooper 42, page 246). For now, it is enough to say that overlapping concepts make an application hard to learn.

Avoiding Blooper 23

This blooper results from perceiving a UI from the designers' perspective—which is biased by knowing what everything in the UI means—rather than from the users' perspective (Basic Principle 3, page 26). The terminology for a

software product should reflect the users' perspective and should be designed to be easy to learn and remember (Basic Principle 6, page 37). Three rules will help you achieve that.

Avoid synonyms

Don't use words that are normally synonyms to mean different things in the software. Make sure the software's terms for its various concepts are clearly distinguishable from one another. Google's GMail application, in contrast with Eudora, uses only one term for searching: "search" (Figure 4.8).

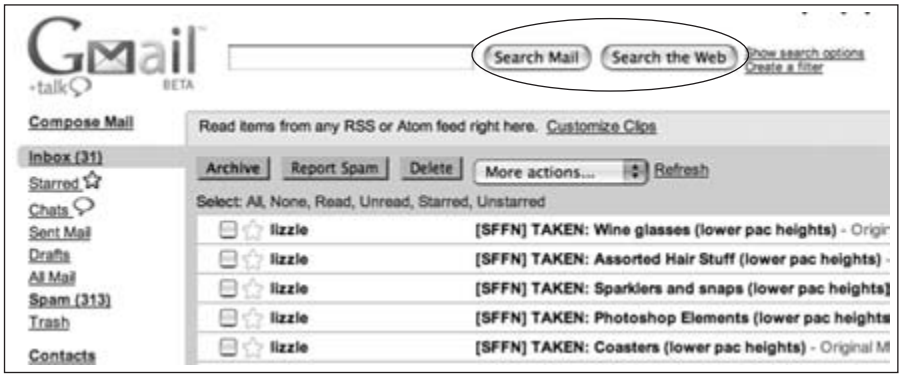
Avoid ambiguous terms

Avoid using terms that are ambiguous in the real world or that have real-world meanings that could be confused with their meanings in the software. Don't assume that just because you've defined a word to have a certain meaning, users will interpret it the same way. Consider how *users* will interpret the words you've chosen.

Test the terminology on users

Software developers sometime say: "That term isn't confusing. It's obvious what it means!" Whether a term is confusing is not for software developers to judge on their own; it must be determined by observing and asking users. Therefore, it is not enough to produce a conceptual model and product lexicon. The lexicon must be tested on representative users, as was explained under how to avoid Blooper 22. If testing identifies terms that users confuse with one another, change them.

Figure 4.8



Gmail: one term for searching: "search."

If users misinterpret the terminology used in your software, it's not their problem; it's *your* problem. They'll use something else that doesn't mislead or confuse them. Therefore, try hard to find terminology that does not mislead or confuse your users.

Blooper 24: Bad writing

Even if software uses terms consistently, doesn't redefine common words, and avoids programmer jargon, code terms, and ambiguous words, the writing can still be inadequate for the commercial marketplace. It can vary in style from one label to another. It can use bad spelling and grammar. It can be incorrectly capitalized. In short, it can be bad writing.

Even if it doesn't hurt usability, poor writing tells customers: "We are amateurs! We don't know how to produce polished products!" This blooper occurs in two variations.

Variation A: Inconsistent writing style

Many applications exhibit stylistic inconsistencies in the text of built-in instructions, command names (in menus and on buttons), setting labels, window titles, and so on. Common inconsistencies include:

- naming some commands after actions (verbs) but others after objects (nouns), e.g., Show Details vs. Properties;
- using terse, "telegraphic" language for some setting labels or messages (e.g., "Enter send date") but wordy language for others (e.g., "Please specify the date on which the message is to be sent");
- using title capitalization (e.g., Database Security) for some headings but sentence capitalization (e.g., Database security) for others;
- ending some but not all sentences (e.g., in instructions or error messages) with periods.

Here are some examples from software I've reviewed:

- In the Startup dialogue box, two choices were labeled "Create New Study" and "Open An Existing Study." Only the second includes the article "An."
- Some fields for entering names were labeled "X Name" while others were labeled "X", e.g., "Table Name:" vs. "File:". Both should include the word "Name" or neither should.
- The Graph menu on the menubar contained the inconsistently capitalized commands: "Add Meter …," "Print meter …," "Add Graph …," and "Print graph …." Probably different programmers implemented the Add and Print functions, and no manager or technical writer checked their command labels for consistency.

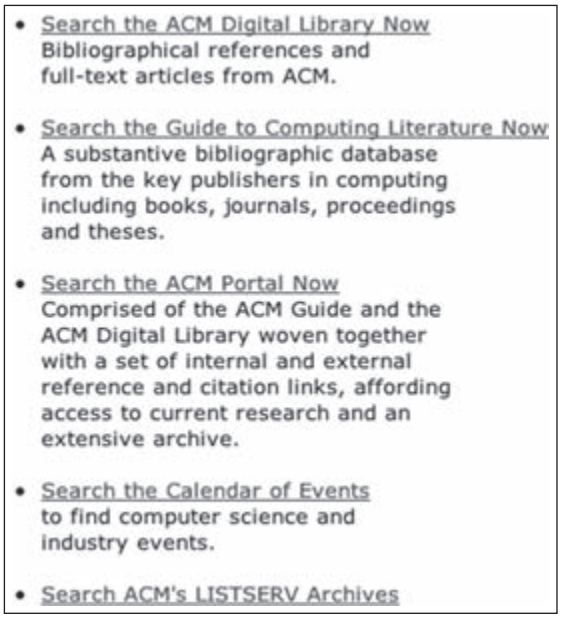
The "Search" page of the Web site of the Association for Computing Machinery (ACM.org) has an example of inconsistent writing style. It offers five different searches. Three say they search "now" (Figure 4.9). Users may wonder if this means the other two search *later*. Three of the links mention ACM. Users could assume this means the others are for other organizations. In fact, all five links are for ACM, and all five search now. The options are labeled inconsistently. This conveys a lack of standards and therefore amateurishness. It can also cause confusion.

Variation B: Poor diction, grammar, spelling, and punctuation

Many software applications suffer from poor spelling and writing. Although user documentation is usually written by technical writers, text that appears in the software is usually written by programmers. Programmers are trained to write code, not prose text, and it shows in the quality of the writing in many programs.

Two costly examples of poor writing in software are provided by two medium-sized Silicon Valley software companies that shall remain nameless. Each had a team developing a large desktop application for the Microsoft Windows operating system. On both projects, the engineers were responsible

Figure 4.9



ACM.org: inconsistent language. Some links include ACM, others don't. Some end with "now"; others don't.

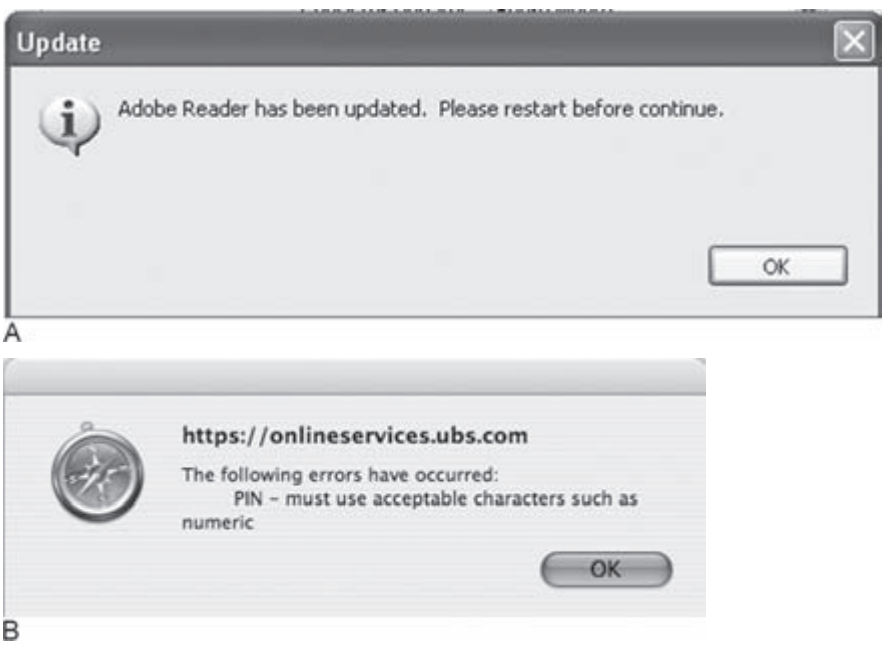
for all text displayed by the software, none of which was reviewed by technical writers. In fact, one of the two teams didn't even have any technical writers.

Although the software was intended for English-speaking customers, none of the developers on either team were native speakers of English. Both of these companies hired most of their programmers from overseas, mainly India, Taiwan, and Russia. While the engineers on both teams were highly competent programmers, their attempts to write command names, setting labels, error messages, and instructions bordered on amusing. However, potential customers were not amused. Management at each of these two companies probably thought that their hiring practices provided skilled programmers at a discount, but they failed to anticipate that those practices would also either add reviewing and rewriting costs or reduce the sales of the product.

Recent examples of poorly written text are provided by Adobe Reader and UBS.com (Figure 4.10). Odds are that neither error message was written by a trained writer of English or even by someone *fluent* in English.

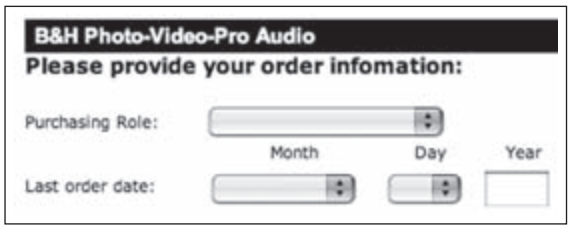
Not all examples of poor writing are serious enough to impair understanding. Sometimes they are just simple typographical errors that were not caught. Typographical errors in software give users an impression of careless workmanship and amateurishness. An example of such an error occurs in the Web site of B&H Photo/Video/Audio. The order form (Figure 4.11) has a typographical error. Can you spot it? It should have been caught before release.

Figure 4.10



Poor English. (A) Adobe Reader. (B) UBS.com.

Figure 4.11



BandHPhotoVideo.com: typographical error in e-commerce Web site.

Avoiding Blooper 24

By following these rules, software developers can ensure that the text displayed by their software conveys an impression of professionalism and care.

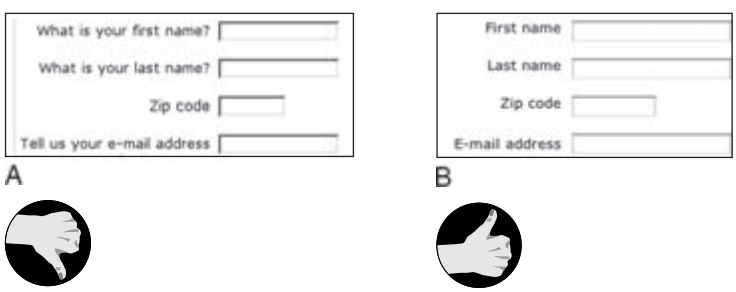
Use people who are skilled at writing

If software developers want their products and services to be professional, they need to get professionals for *all* of the jobs involved with software development. Programmers are professionals at writing code. They are amateurs at writing prose. Programmers should not write the text that appears in software. Information architects and technical writers are the right professionals for that job.

All text in an application—instructions, warnings, error messages, setting labels, button labels—should be reviewed by the information architect, technical editors, and technical writers. Not only does this improve the quality of text displayed by software, it helps ensure consistency with user manuals.

Until recently, the Web site FinancialEngines.com provided a example of inconsistent writing style. The site's customer registration form asked for three pieces of information—name, postal zip code, and e-mail address—using three different label forms: question, one-word label, and command (Figure 4.12A). Recently, the page was revised to correct the inconsistency (Figure 4.12B) and so now conveys a more professional impression.

Figure 4.12



FinancialEngines.com. (A) Inconsistent form labels. (B) Consistent.

Spell-check all text

All text appearing in an application should be spell-checked. The first pass would be with spell-checking software. Ingenuity may be required to get spell-check software to check message files. The text should also be checked by human technical editors or writers.

Blooper 25: Too much text

An important type of bad writing is too much text. Needless text is bad anytime [Strunk and White, 1999], but especially bad in software. When navigating to what they want, software users don't read; they scan for anything that matches their goals [Krug, 2005]. Unfortunately, unnecessary text is very common on the Web and fairly common in desktop applications.

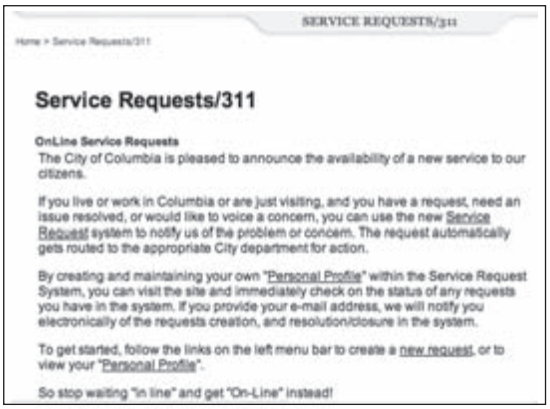
Verbose instructions and labels

Verbose labels and instructions, when not ignored, "bury" important information and slow users down. Imagine yourself trying to set the drawing application SmartDraw's Text Entry Properties (Figure 4.13).

Figure 4.13

SmartDraw: overly wordy instructions and labels.

Figure 4.14



A

B

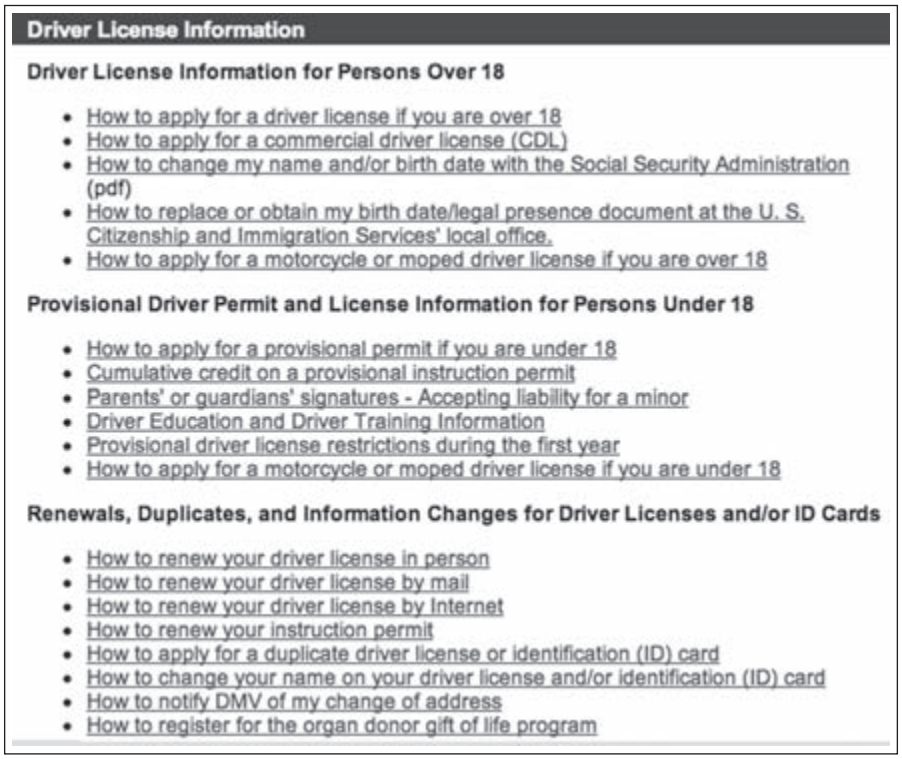
ColumbiaSC.net: wordy instructions. (A) "Service Requests" page. (B) Login/registration page.

Similarly, unneeded "welcome" text and overly wordy instructions just get in the way of users of the Columbia, South Carolina, city government Web site (Figure 4.14).

Lengthy links

Textual links in Web sites, when too long and especially when in lists, are hard to scan. If text is repeated between links, scannability and legibility suffer even more. A long list of links from the California Department of Motor Vehicles' Web site shows this (Figure 4.15). Headings are also too wordy, e.g., the first duplicates "Driver License Information" from the title just above it.

Figure 4.15



Driver License Information

Driver License Information for Persons Over 18

- How to apply for a driver license if you are over 18
- How to apply for a commercial driver license (CDL)
- How to change my name and/or birth date with the Social Security Administration (pdf)
- How to replace or obtain my birth date/legal presence document at the U. S. Citizenship and Immigration Services' local office.
- How to apply for a motorcycle or moped driver license if you are over 18

Provisional Driver Permit and License Information for Persons Under 18

- How to apply for a provisional permit if you are under 18
- Cumulative credit on a provisional instruction permit
- Parents' or guardians' signatures - Accepting liability for a minor
- Driver Education and Driver Training Information
- Provisional driver license restrictions during the first year
- How to apply for a motorcycle or moped driver license if you are under 18

Renewals, Duplicates, and Information Changes for Driver Licenses and/or ID Cards

- How to renew your driver license in person
- How to renew your driver license by mail
- How to renew your driver license by Internet
- How to renew your instruction permit
- How to apply for a duplicate driver license or identification (ID) card
- How to change your name on your driver license and/or identification (ID) card
- How to notify DMV of my change of address
- How to register for the organ donor gift of life program

DMV.ca.gov: wordy and repetitious text links are hard to scan.

Avoiding Blooper 25

Use no more text than is necessary to convey the intended information.

- Avoid long prose paragraphs.
- Use headings, short phrases, bullet points.
- Keep links short, one to three words; explain with nonlink text.
- Avoid repetition in link lists; cut repeated text or move it into headings.

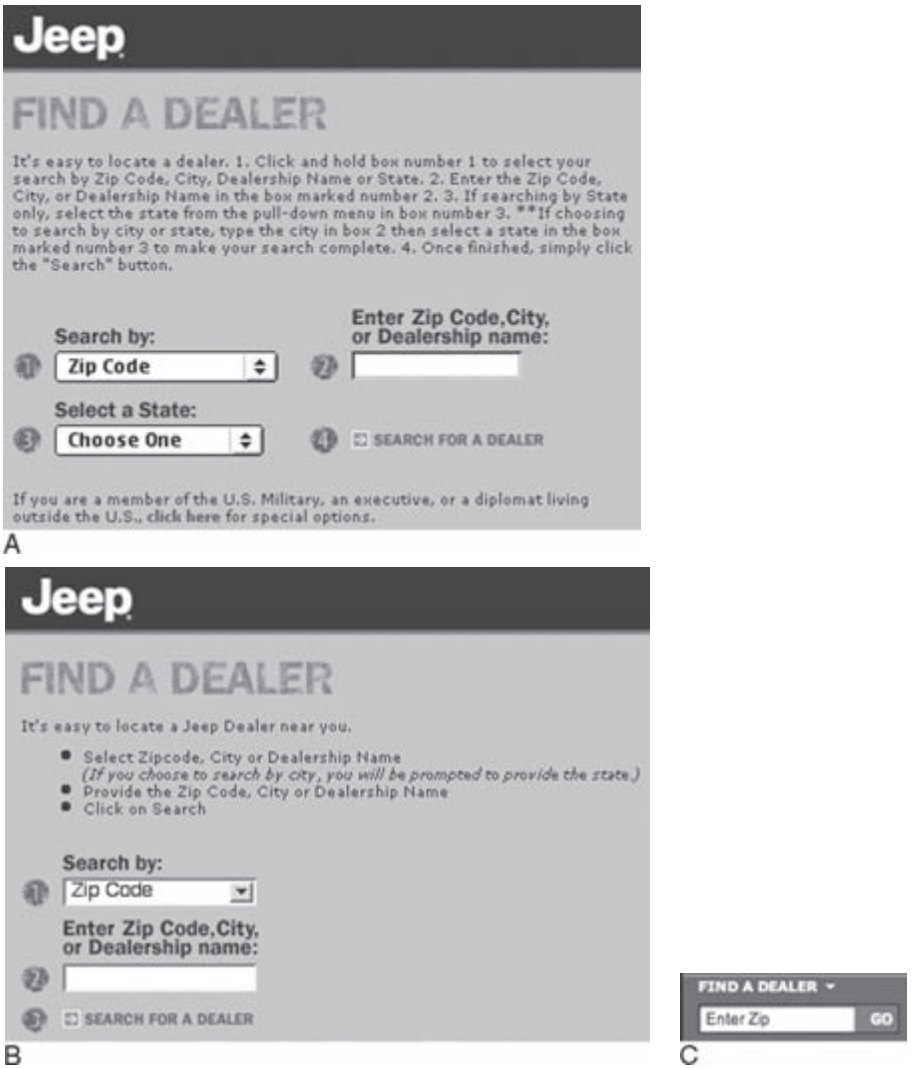
Usability authors Jakob Nielsen [1999d], Steve Krug [2005], and Ginny Redish [2007] all advocate brevity. Krug warns that long prose passages won't be read and suggests:

> Get rid of half of the words on each page, then get rid of half of what's left.

Example of cutting needless text

Jeep.com shows how text can be cut. In late 2002, they simplified their verbose "Find A Dealer" page (Figure 4.16A): a long paragraph was cut to one sentence and three bulleted steps (Figure 4.16B). They also realized they didn't need both zip code *and* state. More recently, they simplified it further, to six words, including the labels (Figure 4.16C).

Figure 4.16



Jeep.com: wordy instructions cut to a few bullets, then to six words. (A) Early 2002. (B) Late 2002. (C) 2007.

The goals: scannability, clarity, simplicity

Brevity is only a means to the true goal: ease of comprehension and navigation, scannability, clarity, simplicity. Users don't read; they scan. Brevity helps them comprehend and navigate by scanning.

If you forget this and strive for brevity for its own sake, usability can suffer [Raskin, 2000]. Needlessly limiting button or link labels to one word can seem to users like cryptic codes they must learn.

Developer-centric text

The next three bloopers are cases of text using computer jargon and presenting the developers' point of view.

Blooper 26: Speaking Geek

Suppose you installed some new software on your computer, but when you tried to use it, you discovered that you had somehow obtained a foreign-language version of the software. You would discard it and try to get the version that was in your own language.

For many people, the "foreign" language their software displays is technobabble, also known as Geek. However, people whose software speaks Geek are *worse* off than those whose software uses a foreign language because they can't get replacement versions in a language they understand. There are several different ways to speak Geek.

BEETLE BAILEY © KING FEATURES SYNDICATE

Variation A: Using programmer jargon

Most professions and hobbies have a jargon—a specialized vocabulary that allows practitioners to communicate more precisely and efficiently. Using specialized jargon is good when you are communicating with others who share your specialty: it enhances efficiency and clarity. However, using specialized jargon when communicating with people who do *not* share the specialty is bad: it *hinders* efficiency and clarity. When communicating with people outside of your area of expertise, you should switch off the jargon.

Many software developers don't switch off their jargon when writing text that appears in software intended for nonprogrammers. Why?

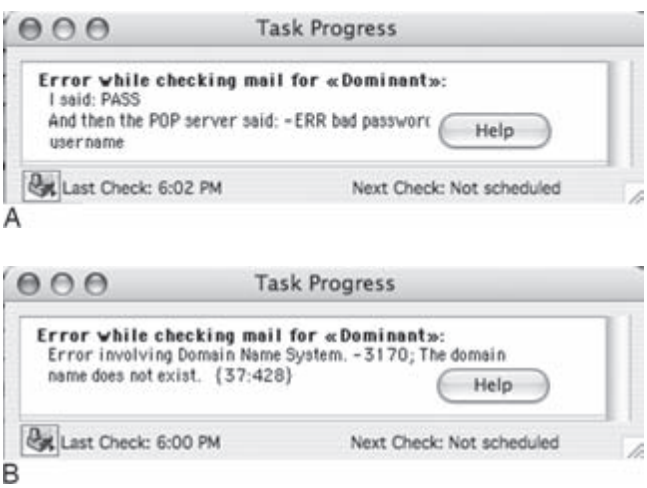
- A lack of awareness that they are using jargon
- An inability to switch the jargon off even though they are aware that they use it
- A belief that if people want to use a computer, they need to understand computer jargon
- A tight deadline and insufficient writer support, coupled with an assumption that someone will check and improve the wording later
- A design that exposes technical concepts and implementation details that are irrelevant to users' tasks (Blooper 40, page 241)

For these reasons, a lot of software uses acronyms such as "USB" and "PDF," pure computerese such as "device drivers" and "flash memory," words that are rarely used in standard English such as "mode" and "buffer," phrases that turn verbs into nouns such as "do a compare" and "finish an edit," and terms that reflect the developer's point of view rather than the user's such as "user defaults." The effect on users is reduced understanding and slowed learning.

Many error messages displayed by the e-mail program Eudora are in Geek. When a user logs in with an invalid password, Eudora pops up an error message that describes the communications protocol between Eudora and the e-mail server (Figure 4.17A). Blooper! Maybe Eudora's developers and network administrators care about the client–server communication, but most Eudora users do not.

Eudora displays an even more geeky error message when an attempt to fetch new mail fails because the domain-name server did not respond (Figure 4.17B). This message is cryptic even for Eudora users who know what a domain-name server is.

Figure 4.17



Eudora: technobabble error messages. (A) Invalid password. (B) Server didn't respond.

Examples of software speaking Geek

- *Example 1:* A Web application required users to login, but called it "authenticating." The login page was labeled "User Authentication." This was bad because: (1) users don't know what "authentication" means and (2) the word "user" is software developer jargon; users don't identify with that term. Worse, if a user left the application unused for more than 15 minutes, the application's back-end logged the user off, but the application in the user's Web browser continued to appear as the user left it. Users who had interrupted their use of the application to do something else would often return to the application, find it as they expected, try to do something, and suddenly get the following message:

  Your session has expired.
  Please reauthenticate.
  [OK]

  When users clicked "OK," they were taken to the "User Authentication" page. I suggested getting rid of the word "user," replacing "authenticate" with "login," and increasing the auto-logoff timeout (since they couldn't eliminate it).

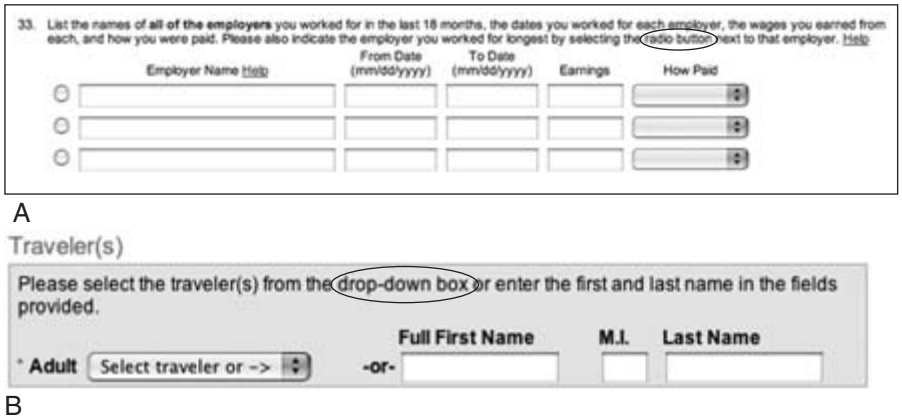
- *Example 2:* A company developed an e-commerce application for networked PCs. The application let users create and save templates for common transactions. It gave users the option of saving templates either on their own PC or on a network server. Templates stored on the server were accessible by other users; templates stored on the PC were private. The problem was that the software referred to the two storage options as "database" and "local." The developers used "database" for templates on the server because the they were kept in a database. The developers used "local" for templates on the users' own PC because that's what "local" meant to them. I recommended that they use "shared" or "public" instead of "database" and "private" instead of "local."

Some software applications and Web sites include the GUI toolkit name of a control in its label or instructions. Two examples come from the Web sites of the State of California's Employment Development Department and Northwest Airlines (Figure 4.18).

One business application displayed the structure of the application's data using a Windows tree control. The problem was that the application called it a "tree control." Not surprisingly, testing showed that many users didn't know what the term meant. Some probably wondered what the software had to do with trees.

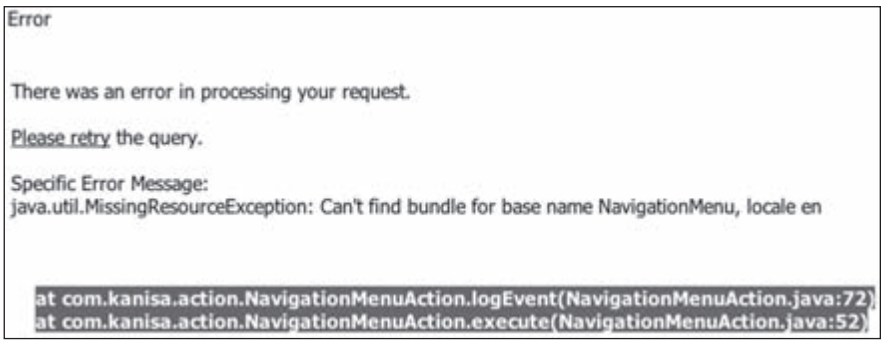
Also common are error messages that contain actual bits of software code. Most computer users have seen error messages like one displayed by Intuit.com (Figure 4.19), which mixes code excerpts with information users can understand: "There was an error in processing your request." The code excerpts may have been useful to programmers while the site was being debugged, but should have been removed before the site went live.

Figure 4.18



Exposing GUI toolkit control names. (A) EDD.ca.gov. (B) NWA.com.

Figure 4.19



Intuit.com: exposes code in error message.

Sometimes users are exposed to implementation terms because the operating system displays error messages on its own instead of passing them to the application. For a more complete discussion of such situations, see Blooper 28 (page 184).

Variation B: Turning common words into programmer jargon

Programmers often redefine common words to have specific meanings in software. If you redefine common words and expect the users to adapt to them, you are putting an extra load onto the users and committing a blooper.

In common English "dialog" means a conversation, but in GUI-programmer jargon it is shorthand for "dialog box." Programmers and UI designers

forget that "dialog" has a common meaning and that, even in GUI jargon, it is shorthand. When software exposes the jargon meaning to users, as in Adobe InDesign (Figure 4.20), it's a blooper.

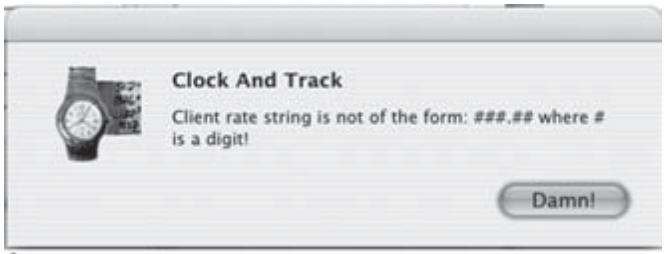
To programmers, the word "string" means textual data in software. To nonprogrammers, string is for tying things together. Examples of exposing the software jargon meaning to users come from the desktop software Clock and Track and the publisher Web site Elsevier.com (Figure 4.21). This use of "string" should *never* appear in a UI intended for nonprogrammers.

To most English speakers, an argument is a verbal dispute. To programmers, arguments are input to software functions. Some of CorporateExpress.com's users might like to have an argument with the designers of the site's Search function (Figure 4.22).

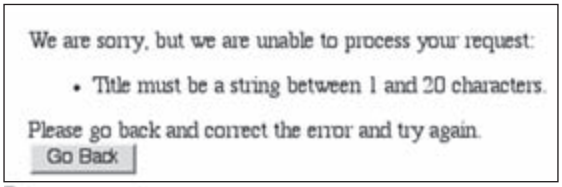
Figure 4.20

Adobe InDesign: exposes the GUI jargon term "dialog."

Figure 4.21



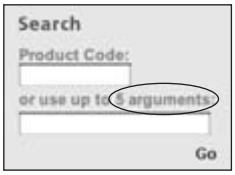
A



B

Error messages expose the software term "string." (A) Clock and Track application. (B) Elsevier.com.

Figure 4.22



CorporateExpress.com: exposes the software jargon word "arguments" in instructions.

She did what it said

A secretary called the Compuserve customer support hotline to say that even though she did what the software told her to do, it didn't seem to work. Compuserve's software had displayed an error dialog box containing the message:

Type mismatch

The secretary said that when she saw this message, she typed "mismatch" several times, but it didn't help. [From Interface Hall of Shame, http://homepage.mac.com/bradster/iarchitect/shame.htm]

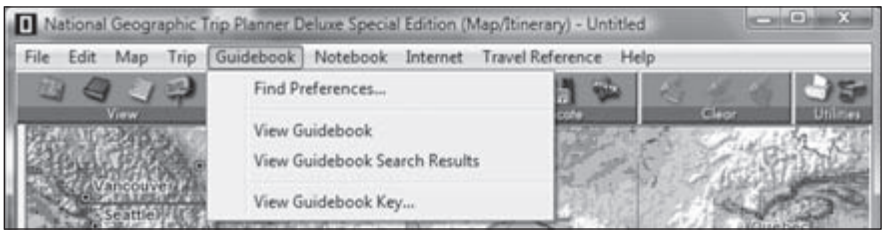
Other often-redefined words are "resources," "object," and "client." One Web application had a login page titled "Thin-Client Login." Some users were probably surprised and pleased that it offered them the login page for *thin* clients instead of the alternative.

Variation C: Turning verbs into nouns

Another sort of jargon is verbs used as nouns. This is not restricted to computer software; it happens in many fields. Stockbrokers use "buys" and "sells" as nouns when discussing stock transactions, airplane pilots refer to "takeoffs," fishermen talk about the day's "catch," and book reviewers describe books they like as "a worthwhile read." Programmers say "the compile failed," "start the build," "do a compare," "finish an edit." This is fine when communicating with other programmers, but bad when communicating with nonprogrammers.

National Geographic Trip Planner includes a guidebook with information about routes and destinations. The guidebook has a Find function so users can search it. Users can set preferences for how Find works. The command to do that is Find Preferences (Figure 4.23).

Figure 4.23



National Geographic Trip Planner: Find Preferences command is a noun phrase.

Verbs to nouns

Here are two more examples of programmers turning verbs into nouns:

- In a data-mining application, one function was called "Explore Data." The programmers called using that function "doing an Explore." An auxiliary function predicted the time and resources needed to "do an Explore." It was named "Explore Prediction," a noun phrase. Another function compared two data files. Using it was called "doing a Compare." An auxiliary function defined comparisons that could be run repeatedly on different data. It was called "Compare Definition," another noun phrase.
- An application provided a "wizard" (multistep dialog box) for creating data objects. The button that started this wizard was labeled "Create Object Wizard." The developers meant for this label to mean "start the (Create Object) wizard." However, users read it as a verb phrase, "create the (Object) wizard," and wondered what an "Object" wizard was and why they would want to create one.

Avoiding Blooper 26

Geek-speak must go.

When commercial automobiles were first introduced and for about 40 years afterward, operating one required mastering a lot of automotive technical jargon: choke, RPMs, oil pressure, generator voltage. Now most of that jargon is gone. Computer applications began to appear in the late 1970s. Thirty years have passed since then. In 10 more years, will your software's users still have to be aware of modems, startup files, device drivers, and RAM? Hopefully not. Let's not wait 10 more years; let's achieve that goal now.

How can developers avoid Geek-speak? By following these steps.

Know thy users

Learn about your users. Visit them, observe them, interview them, invite them to focus groups. Ask them to describe how they work, what they like and don't like about their current tools, and what their most serious problems are. Get their ideas about how their work might be improved. Compile a list of all the concepts the intended users mentioned in their descriptions of their work. Pay special attention to the objects (nouns), actions on objects (verbs), and attributes of objects (adjectives) they mention.

Develop a product lexicon based on users' task vocabulary

Use the information gathered from intended users to develop a conceptual model for the planned software product or service (Basic Principle 2, page 18).

The conceptual model will include an object/action analysis and a lexicon. The lexicon should list all the concepts (objects, actions, attributes) that the software exposes to users and indicate the name for each concept. When possible, use industry-standard names.

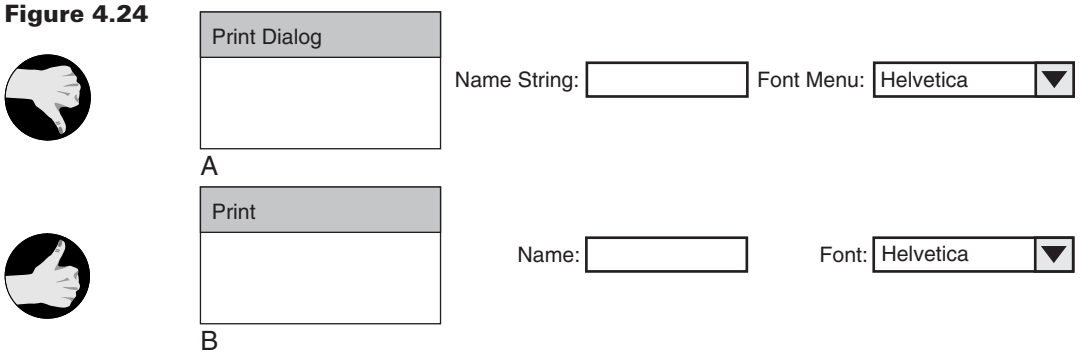
The goal is that the user interface and documentation use a vocabulary that is consistent, both internally to the software and with industry standards for the platform, and also grounded firmly in the tasks and vocabulary of the intended users (Basic Principle 3, page 26). Toward that end, develop and maintain a product lexicon and enforce adherence to it. All text displayed in the software should be written or at least reviewed by a technical writer, and any Geek-speak should be filtered out. Labels and messages should be in message files, separated from the program code, to facilitate review and translation.

Some have argued that the UI and the terms it uses should match the implementation, so the UI does not mislead users about how the application works. Yes, but the right way to accomplish that is to design the UI first and then match the implementation—structure, concepts, and terminology—to that.

Leave GUI component names out of the GUI

Don't include the GUI toolkit name of controls in the title or label for a control. Figure 4.24A shows examples of geeky labels: they include GUI toolkit jargon. Figure 4.24B shows good labels for the same settings. The good labels eliminate needless words as well as Geek-speak.

Figure 4.24



(A) Labels that include GUI toolkit jargon. (B) Improved labels.

Blooper 27: Calling users "user" to their face

Related to speaking Geek is the blooper of callin g your users "user" in the UI.

"Users" is what we—software and Web *developers*—call people who use our systems. It's a *fine* term to use when we are talking with other designers and developers. It's part of our professional jargon—our way of communicating succinctly with peers. But "users" is *not* what people who use computer-based products and services call themselves.

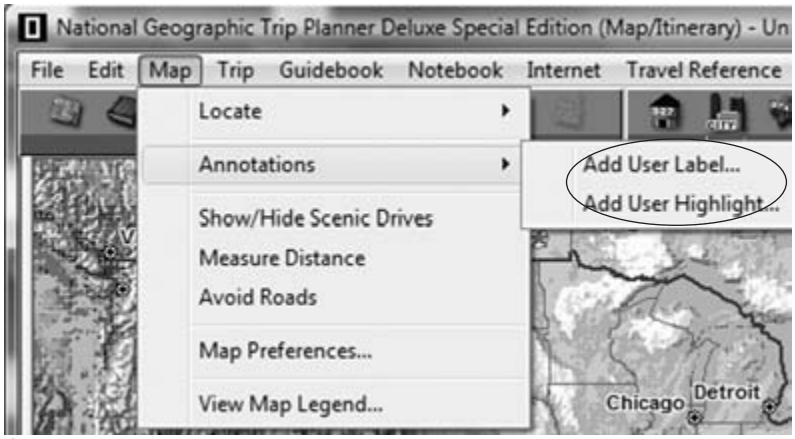
Only two industries call their customers "users." One such industry is ours: computer software. Do you know what the other industry is?[3]

Software applications, Web sites, and electronic devices should be designed from the point of view of the people who *use* them, not the point of view of the system's designers or developers. The people who use computer-based products and services see themselves as customers, site visitors, members, guests, etc.—not "users." Therefore, interactive systems that call users "user" to their face are committing a blooper.

National Geographic Trip Planner allows users to annotate and highlight locations on the program's maps that interest them, but uses the terms "User Label" and "User Highlight" (Figure 4.25A). Microsoft Windows XP's "Accessibility" control panel exhibits two points of view simultaneously: it calls a user-specified style sheet "User style sheet" in one label and "my style sheet" in another (Figure 4.25B).

FinancialEngines.com and LinkedIn.com not only call users "user"; they make users call *themselves* that (Figure 4.26).

Figure 4.25



A

Microsoft calling users "user." (A) National Geographic Trip Planner. (B) Windows XP.

(Continued)

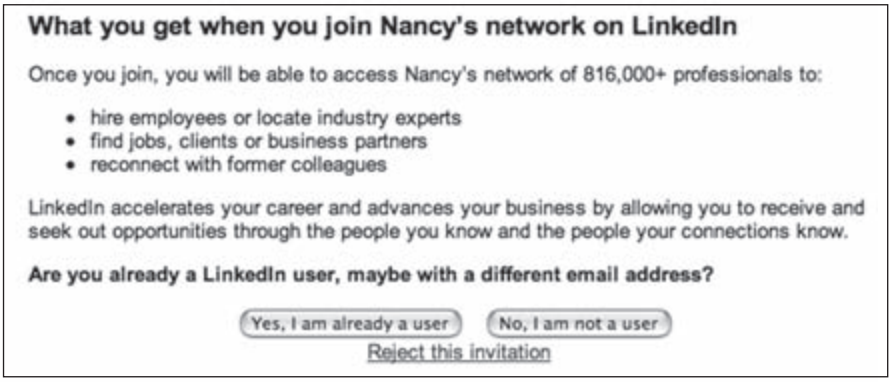
3. Hint: heroin, cocaine, methamphetamines.

Figure 4.25
(Continued)

B

Figure 4.26

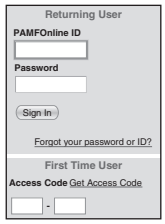
A



B

Making users call themselves "user." (A) FinancialEngines.com. (B) LinkedIn.com.

Figure 4.27



PAMF.org: buttons on "Accept/Reject Link" page force users to call themselves "user."

A potentially ambiguous case occurs at PAMF.org (Figure 4.27), a medical clinic. In that context, "user" could be interpreted to mean "drug user."

Calling users "user" to their face is an easy mistake to make: "user" is developers' jargon for people who use the Web site. If a development team doesn't explicitly *think* about this and choose a more appropriate word, "user" is the word that will be used. That's why this blooper is so common.

Avoiding Blooper 27

As easy as it is to make this blooper, it's just as easy to avoid it. Using a non-developer-centric term like "visitor," "customer," or "member" instead of "user" costs next to nothing. It just takes awareness and a few moments' thought. Three Web sites that show evidence of such awareness and thought are Yale AlumniConnections.org, Fedex.com, and Apple Mac.com (Figure 4.28).

Figure 4.28

Not calling users "user." (A) Yale AlumniConnections.org. (B) Fedex.com. (C) Apple Mac.com.

Blooper 28: Vague error messages

A blooper related to speaking Geek is displaying error messages that announce a vague, generic error condition instead of giving users helpful, task-oriented information about what happened and what to do about it. This happens for three reasons.

Variation A: Message displayed by low-level code

Sometimes low-level service functions detect errors and display error messages themselves. Task-level functions the user explicitly executes—e.g., Save—can express errors in task-related terms, but low-level service functions—e.g., file.Open()—cannot.

In Apple's Safari Web browser, suppose a user tries to visit a florist's Web page. If the page has buggy JavaScript code that tries to assign a null value to a variable, Safari's JavaScript interpreter displays an error: "TypeError—Null value" (Figure 4.29).

The user's reaction would probably be something like: "Huh? I just want to send some flowers to my mom. What's this about JavaScript, type errors, and null values?" If the user is more computer savvy, she might say: "You stupid browser—I didn't write the faulty JavaScript code. Show the error to the site's *developers,* not to its *users!*"

Error message displayed by low-level code

A top-notch programmer had emigrated to the United States. His poor English was not a problem because he wrote low-level device-driver code that had no UI. He was asked to write a driver for a new display. No one checked his work because the driver was supposedly invisible to users. And it was … almost. It had a bug that occasionally caused it to hit a memory limit and display this message:

Nesting level too dip.

This error message and the code that displayed it had already been burned onto ROM and shipped with thousands of consoles worldwide. The main problem with the error message was *not* the misspelling of the word "deep," but that it was displayed by the display's firmware. Users of the display would have no idea what the message was about or what to do about it.

Figure 4.29

Apple Safari browser: error message from JavaScript interpreter.

This variation of Blooper 27 is often difficult to correct. The low-level code that detects the error and displays an unhelpful error message may not be in the application itself, but in the operating system on which the application is running. If your application calls an operating system utility function and the function sometimes displays a poor error message, you probably won't be able to fix the message because it isn't from your code. Nonetheless, your users will see the message as coming from your application. If the message is seriously misleading, you have no choice other than changing the code so that it does not use the operating system utility function.

Variation B: Reason for error not given to higher level code

Sometimes applications display vague error messages because of poor communication between low-level service functions and the task-level function the user executed. For example, when a user tries to open a PowerPoint presentation but PowerPoint can't open it, an error message pops up listing three possible reasons for the failure (Figure 4.30). The service functions PowerPoint calls to open and load the presentation apparently don't give PowerPoint enough detail about why they failed to allow it to identify the exact cause of the failure.

Users are left not knowing what to do, because the remedies for these three possible causes are quite different.

One application displayed this error message when a user tried to load a nonexistent data file:

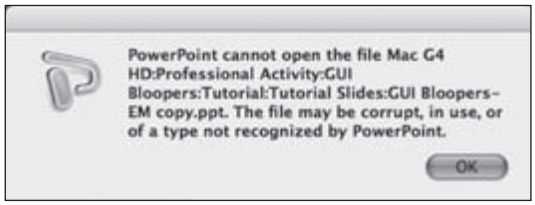
> Error parsing datafile. Data not parsed.

The message was true—no data was parsed—but it was misleading. The real problem was that the data file was not found. After trying to load the file, the code did not check whether the load operation succeeded; it just passed an empty data buffer to the parsing procedure, which duly reported that it couldn't parse the data.

Variation C: Generic message components

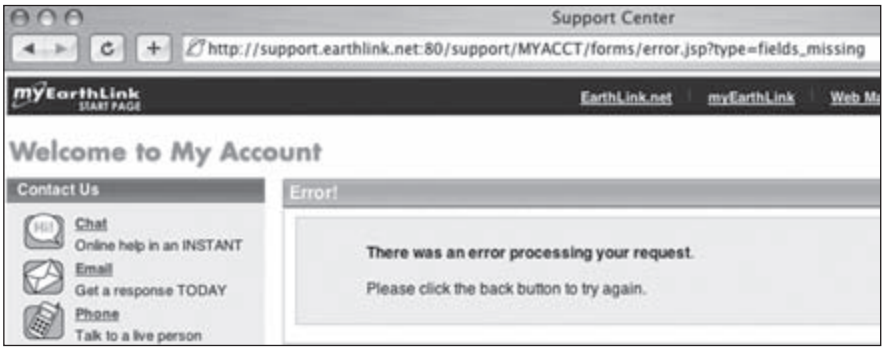
Another common cause of vague error messages is generic error message components. To save development effort, developers sometimes create generic messages to cover whole categories of errors and use them even when the software could give users more specific feedback.

Figure 4.30



Microsoft PowerPoint: error message lists three possible problems.

Figure 4.31



Earthlink.net: vague error message. Specifics in URL, where few users will look.

Earthlink.net displays a vague error message when a user updates his/her contact information but omits a phone number (Figure 4.31). It actually "knows" that the problem is a missing phone number; you can see that in the URL—but normal consumer users won't look there. Yet the error message it displays is generic.

One stock investment application allows investors to buy or sell stocks. Users often get this error message when they place orders:

> Order size not multiple of trading unit.

The number of shares bought or sold must be a multiple of the "trading unit." The trading unit differs for each stock, but the error message does not say what the trading unit is. The trading unit is also not shown anywhere on the ordering screen. Users just have to know, or guess, what the trading unit is for the stock they want to buy or sell. Users not only waste time trying to find the trading unit for a stock, they sometimes start transactions they don't want.

Examples of generic error messages

The following are examples of software that displayed generic, uninformative error messages.

- *Example 1:* User tried to give a data object a name containing characters not allowed in names:

  Name contains invalid characters.

  Wonderful. Pray tell, which characters might those be? The software knows, but won't say.
- *Example 2:* Error message displayed when user tried to import a data file:

  File missing or you don't have access.

  This message is vague: it doesn't say which of two quite different problems has occurred. The message doesn't name the file it is talking about.

Figure 4.32

Windows Media Player (for Mac): vague error message.

And the winner is . . .

The mother of all generic, uninformative error messages—the message that, if there were a prize for vagueness, would be the winner—has to be the error message displayed by Windows Media Player for Mac (Figure 4.32). What are users supposed to do with this?

Avoiding Blooper 28

Express the error in terms of the task

A good error message describes the problem in terms related to the task the user was trying to do. If the user has just given the command to paste an image into a document and the software encounters an error, the message should be expressed in terms of pasting images, not in terms of operating system functions, implementation data types, program exception codes, or irrelevant application concepts.

Don't just identify the problem; suggest a solution

A good error message also provides enough information that the user can see how to correct the error. That means providing enough detail that a user can determine what he or she did to cause the problem or, if the problem wasn't the user's fault, what did cause it and why. A programmer friend of mine put it this way:

> "Error messages should focus on the solution. Geeks love to describe the problem."

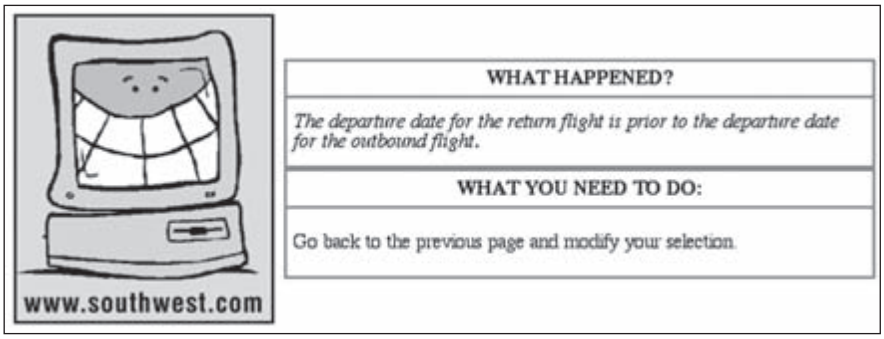
To counter that tendency, error messages should always contain the following:

Error symbol; Problem: Solution

An example of software that follows this guideline is SWA.com, the Web site of Southwest Airlines (Figure 4.33).

Table 4.1 shows suggested improvements for poor error messages discussed above.

Figure 4.33



SWA.com (Southwest Airlines): excellent error message.

Table 4.1 Poor and improved error messages

Poor error message	*Improved error message*
File.Move() failed. [*User tried to save e-mail message, but specified Sent folder as destination by mistake.*]	Messages cannot be saved into the Sent folder. Please try Saving to a different folder.
JavaScript TypeError: Null value	Warning: Page contains coding errors. Content may not display as intended [*or no message*].
Nesting level too dip.	[*No message. Console just restarts if necessary.*]
Name contains invalid characters.	< *Object-type* > names may not contain '-', '/', '@', '#', or '&' characters.
File missing or you don't have access.	[*Separate messages for the two error types:*] File < *filename* > not found. You don't have access to file < *filename* > .
Order size not multiple of trading unit.	Sorry: < *stockname* > stock must be traded in multiples of < *trading unit* > shares.

Pass errors up to code that can translate them for users

Low-level service routines and application software platforms should *never* display error messages directly. They should always pass errors to the application so that it can handle them in an appropriate way.

When an application receives an error notification from a lower level procedure, it should pass the error up the call stack to code that can handle the error in a task-relevant way. That code should either:

- translate the error into terms that are meaningful to users and display the translation with advice on how to correct the problem or
- assume that the cause of the error was temporary and try the operation again.

Design messages and message-bearing components to accept details at runtime

Error messages and error dialog box components that cover many situations should be designed to allow details—object names, constraints, data field names, etc.—to be inserted into them. This would, for example, allow an error message to say *which* file could not be read, show *which* characters are not allowed, or indicate *which* required data was not given.

Different types of messages have different audiences

Finally, recognize that error messages displayed by software have three possible functions, each having a different audience:

- Indicating user errors: for end users
- Logging activity: for system administrators at users' site
- Facilitating debugging and tracing: for developers

By the time software is ready to ship, developers should make sure that each type of message is seen only by its intended audience.

Misleading text

The last three textual bloopers concern error messages and labels that mislead users.

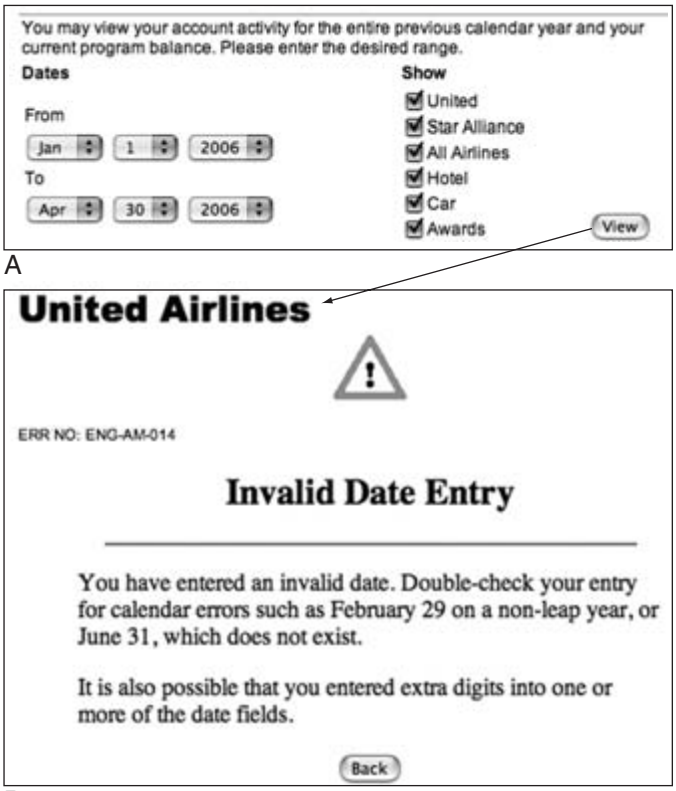
Blooper 29: Erroneous messages

Nothing confuses and angers software users more than instructions and error messages that are wrong. They waste time and effort by leading users down wrong paths, perhaps leading to costly mistakes.

United Airline customers, when reviewing their frequent-flier mileage account on United.com, can ask the site to list account activity that occurred between specified dates. If a user specifies a date that is in the future, the site *could* gently say that, or just overlook the error and use *today* as the upper date limit. Instead, United.com harshly informs users that they have made an "Invalid Date Entry" and tells them to check whether they entered nonexistent dates, such as June 31 (Figure 4.34). The suggested remedy is unrelated to the user's actual error.

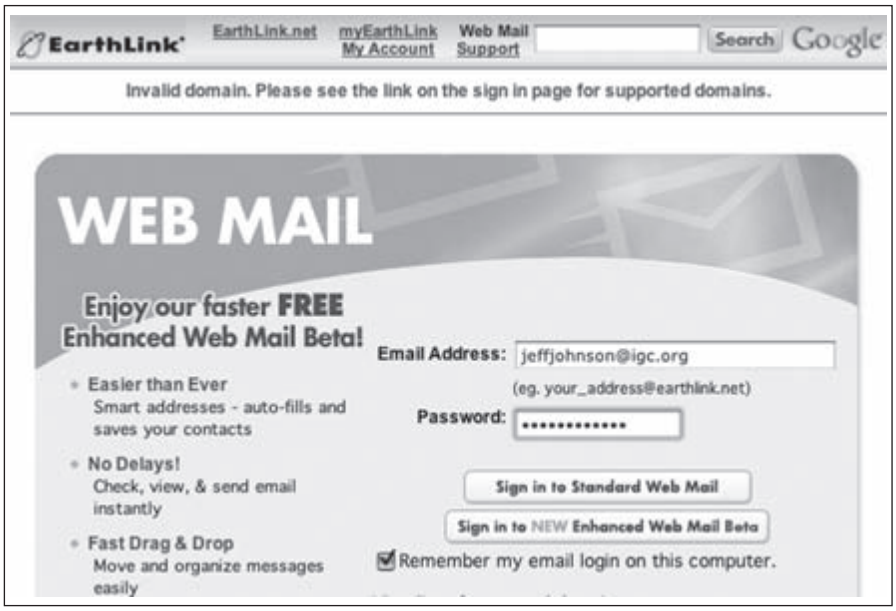
Worse are false error messages when a user hasn't even made an error. If an Earthlink Web-mail customer tries to login at a time when the company's domain-name servers are down, Earthlink Web mail displays an error message telling the user that the specified domain name is "invalid" (Figure 4.35). The domain name is not invalid; the server just cannot authenticate *any* domain names at the moment. This message will cause Earthlink customers to waste time checking and retyping their e-mail address in vain.

Figure 4.34



United.com: incorrect error message. User entered future date, but message misleads.

Figure 4.35



Earthlink.net: incorrect error message. Domain name is valid, but name server is down.

Even worse are error messages that scare users needlessly by announcing something awful when nothing is wrong. Microsoft Windows for the Pocket PC sometimes inexplicably displays a truly frightening system error message (Figure 4.36). The error dialog box commits another blooper as well, trapping users by presenting unclear options (Blooper 50, page 281), but this message's main fault is that it is wrong. Regardless of whether the user clicks Yes or No, the Pocket PC continues operating normally and nothing is erased.

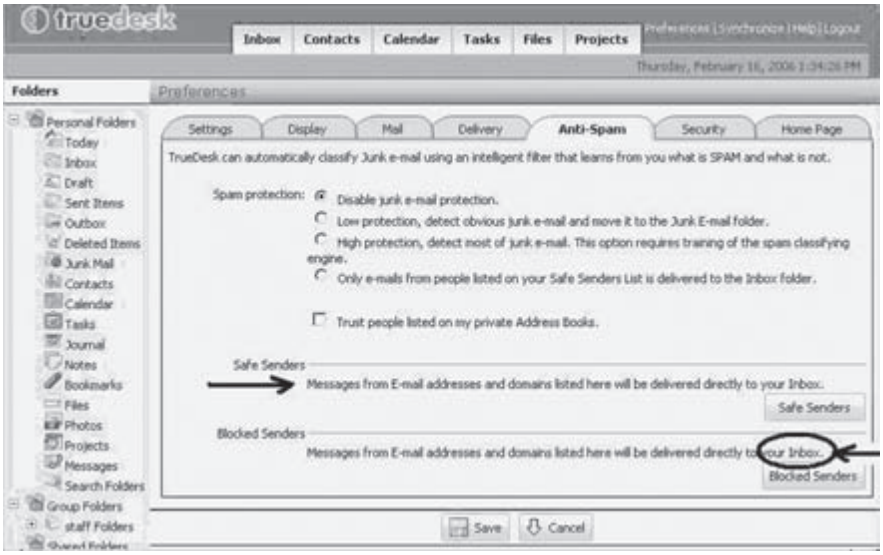
Finally, we have text that is wrong due to carelessness. This problem can often be seen in Web sites in which prices, events, or product catalogues are not kept up to date or in other ways don't match reality.

Erroneous text can also be found in desktop software. The Web-based e-mail program TrueDesk allows its users to create lists of "Safe Senders"—e-mail addresses from which e-mail is trusted—and "Blocked Senders"—e-mail addresses from which e-mail is blocked. However, the "Blocked Senders" description says that blocked e-mail will go exactly where trusted e-mail goes: into the users' Inbox (Figure 4.37). This must be wrong. Whoever created this page apparently copied the text from the "Safe Senders" description to that of the "Blocked Senders" and failed to edit it. But users may not figure this out right away.

Figure 4.36

Microsoft Pocket PC Windows: false error message. Nothing is wrong.

Figure 4.37



TrueDesk: Anti-Spam tab has labels that are wrong.

Avoiding Blooper 29

It is very bad for software to lie to its users. It can waste users' valuable time and effort as well as cause them to make unrecoverable errors.

Error messages that scold users for the wrong error or for errors they did not commit, and instructions that are false, are software flaws—bugs. They should be checked for during software quality-assurance testing and reported and tracked with bug management mechanisms. They should have a high priority for correction, because their impact on users is high: they divert users from achieving their goals, they sometimes cause data loss (or, in the case of mission-critical systems, other types of loss), and they really decrease customer satisfaction.

Blooper 30: Text makes sense in isolation but is misleading in the GUI

Jakob Nielsen has pointed out (at UseIt.com) that software and Web developers often write labels, headings, descriptions, and instructions without considering how users might interpret the text in the context of all the other information the system displays. This often results in text that may make sense in isolation but isn't as clear in the GUI.

Most Web shoppers have been stymied by e-commerce Web sites that display similar descriptions for different items. Imagine a printer vendor Web site on which four different printers are described as "perfect for your small business" or an online catalogue of PhotoShop plug-in filters that all promise to "help you create professional-looking images." The marketing manager in charge of each product naturally wants to make it sound as appealing as possible, but that can make it difficult for customers to choose.

Just as unplanned *similarity* between item labels can sow confusion, so can unintended *differences*. One company's customer support Web site included a page of software patches that customers could download and install to correct known bugs in the company's software. A section of the "Patch" page highlighted patches that the company was strongly recommending that customers install. That section was labeled:

> Recommended Patches
>
> These patches have been tested and will keep your Company X workstation running smoothly.

This might suggest to some customers that the other patches had *not* been tested and were *not* recommended. The person who wrote the section label had considered only how the label fit its own section, not what it implied about the rest of the page on which it appeared.

Avoiding Blooper 30

When writing text describing an item, consider how people who aren't intimately familiar with the item will interpret it. Also, don't simply consider each piece of text in isolation. Look at it in all contexts where it will appear, and make sure it conveys its intended meaning in each such place. When in doubt, test it on users.

Blooper 31: Misuse (or nonuse) of "…" on command labels

In the early 1980s, the designers of Apple's Lisa computer (a predecessor of the Macintosh) devised a way to distinguish commands that execute immediately from ones that first prompt for more information. Commands that need more

information had "…" (ellipsis) on the end of the command label, for example, "Save As…" Commands that execute immediately don't end with "…". This convention is for command labels, whether they appear in menus or on buttons.

The ellipsis became a standard

It is helpful for users to know in advance whether a command executes immediately or prompts for more information. It is safer to click on unfamiliar commands if users know they just bring up a dialog box.

Over time, the convention spread beyond the Macintosh to other computer platforms, such as Microsoft Windows and various Unix-based desktop operating systems. Today, it is so pervasive that software not following this convention risks misleading users.

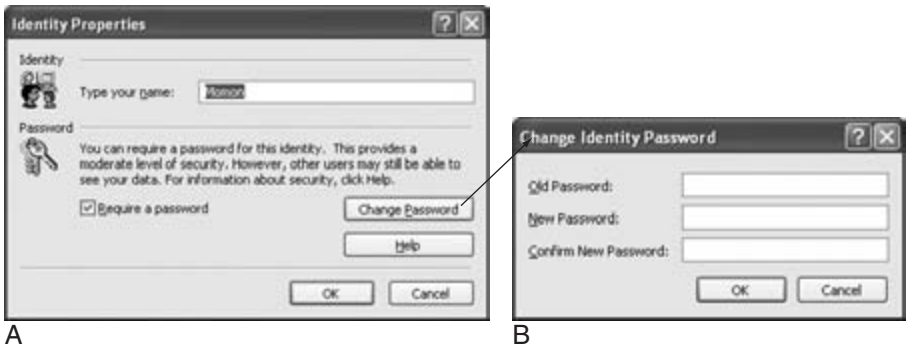
Some developers don't know the standard

Alas, violations of this convention are becoming *more* common. Some developers are just unaware of it. Others know there is a convention but misunderstand it.

Variation A: Omitting "…"

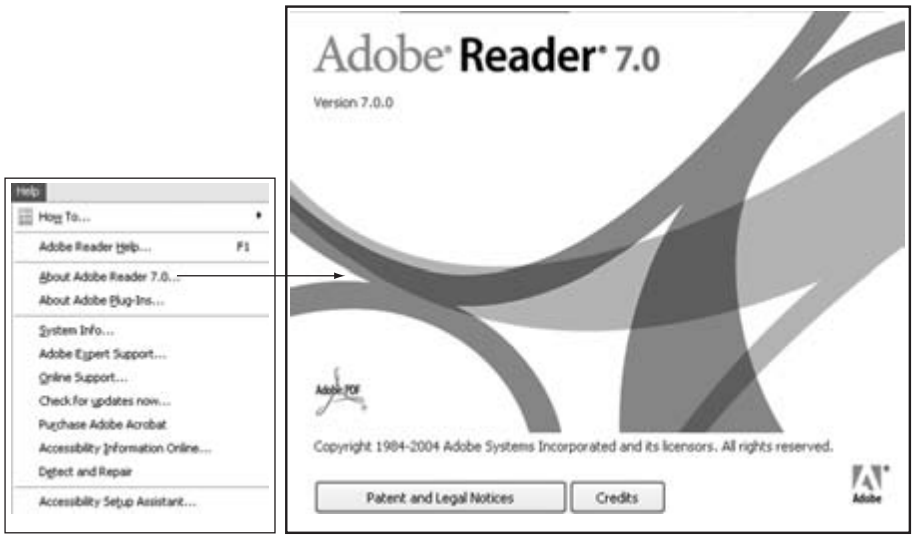
The most common error is to omit the "…" on commands that need it: no commands have "…". Users guess or learn from experience which commands need more input and which don't. Microsoft Outlook's "Change Password" button displays a dialog box to check the user's current password and get a new one (Figure 4.38). The button's label should end with "…", but does not.

Figure 4.38



Microsoft Outlook: missing "..." on "Change Password" button.

Figure 4.39



Adobe Reader: About Adobe Reader 7.0... command wrongly includes "..."

Variation B: Overusing "..."

The next most common error is to append the ellipsis to command labels that should not have it. Designers who commit this variation of the blooper have overgeneralized the convention. They think "..." is for any command that opens a new window. If a Show Graph... command just displays a graph and doesn't need more information before it does that, the command label should not end with "...".

In Adobe Reader's Help menu, the About Adobe Reader 7.0... command displays a splash screen showing the program version and other information (Figure 4.39). It needs no additional input from the user. The "..." is misleading.

Avoiding Blooper 31

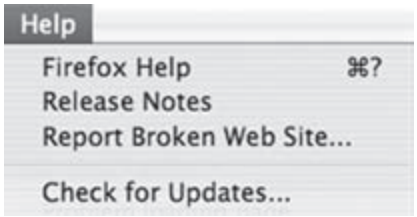
The ellipsis shows that the command brings up a dialog box before executing. It shows that there will be an opportunity to cancel.

Not for any command that opens a window

The "..." is not for commands that just open a window. For example, Show Network might open a window to display the status of a computer network. Such a command should not have "..." at the end of its label.

Mozilla Firefox uses "..." correctly in its Help menu and elsewhere. The Help and Release Notes commands open windows that display information

Figure 4.40



Mozilla Firefox: correct use—and nonuse—of "...".

and so do not end with "…". The Report Broken Web Site… and Check for Updates… commands open dialog boxes that prompt for user input needed to complete the command and so do end with "…" (Figure 4.40).

Standard across all platforms

All the major GUI platform style guides state the same rule:

- Java Look and Feel Guidelines [Sun Microsystems, 2001]
- Windows Vista User Experience Guidelines [Microsoft Corp., 2006]
- Apple Human Interface Guidelines [Apple Computer, 2006]

What about graphical button labels?

Many buttons are labeled graphically rather than textually. The ellipsis mark doesn't work for graphical labels. A solution is to include the ellipsis on the button's tooltip text, which appears when the screen pointer is held over the button.