Safety

Human Error

Error Types

- Slips and lapses
  - Failure to correctly execute a procedure
  - Slip is a failure of execution, lapse is a failure of memory
  - Typically found in skilled behavior
- Mistakes
  - Using wrong procedure for the goal
  - Typically found in rule-based behavior or problem-solving behavior

Errors can be classified into **slips and lapses** and **mistakes** according to how they occur.

Slips and lapses are found in skilled behavior - execution of procedures that the user has already learned. For example, pressing an onscreen button - moving the mouse pointer over it, pressing the mouse button, releasing the mouse button - is a skill-based procedure for virtually any computer user. An error in executing this procedure, like clicking before the mouse pointer is over the button, is a slip. This is just a low-level example, of course. We have many higher-level, learned procedures too - attaching a file to an email, submitting a search to Google, drawing a rectangle in a paint program, etc. An error in execution of any learned procedure would be a slip.

Slips are distinguished from lapses by the source of the failure. A slip is a failure of execution or control - for example, substituting one action for another one in the procedure. **A lapse** is a failure of memory - for example, forgetting the overall goal, or forgetting where you are in the procedure.

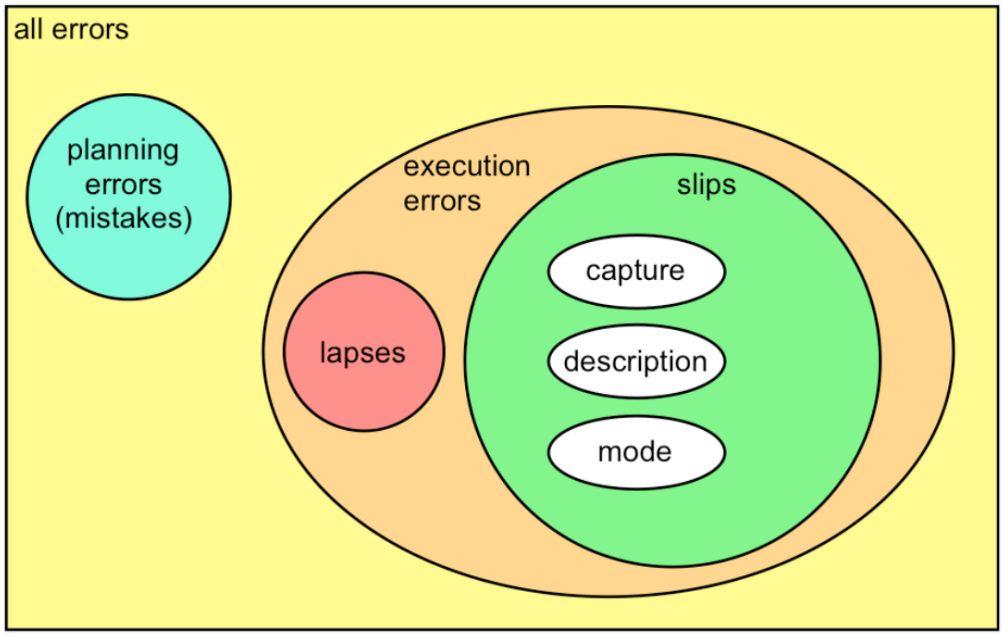
A mistake, on the other hand, is an error made in planning or rule application. One framework for classifying cognitive behavior divides behavior into skill-based (learned procedures), rule-based (application of learned if-then rules), and knowledge-based (problem solving, logic, experimentation, etc.) Mistakes are errors in rule-based or knowledge-based behavior; e.g., applying a rule in a situation where it shouldn't apply, or using faulty reasoning.

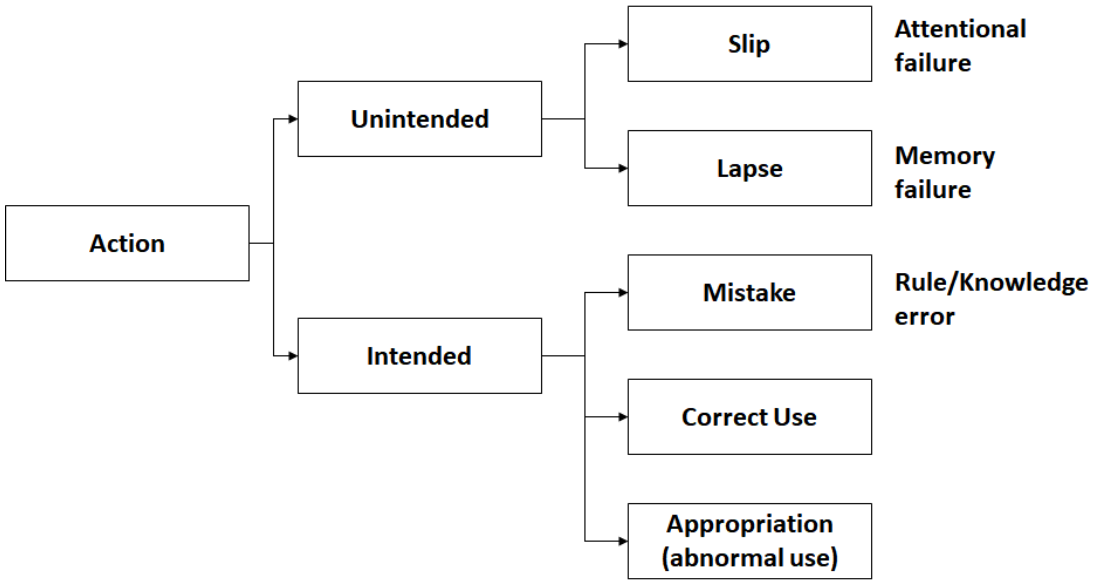
Overall, **slips and lapses** are more common than **mistakes**, because we spend most of our actual time executing learned procedures. If we spent most of our time problem-solving, we'd never get much done, because problem solving is such a slow, cognitively intensive, serial process. I've seen statistics that suggest that 60% of all errors are slips or lapses, but that's highly dependent on context. Relative to their task, however, slips and lapses are less common than mistakes. That is, the chance that you'll err executing any given step of a learned procedure is small–typically 1-5%, although that's context dependent as well. The chance that you'll err in any given step of rule-based or problem-solving behavior is much higher.

We won't have much to say about mistakes in this reading, but much research in human error is concerned with this level - e.g., suboptimal or even irrational heuristics that people use for decision making and planning. A great reference about this is James Reason, Human Error, Cambridge University Press, 1990. Also, you can find the taxonomy in Categorization of Action Slips, Donald A. Norman, 1981.

There are behaviors that are difficult to successfully perform (e.g., putting a thread into the eye of a needle or adding numbers in the head). In this case, let's us just simply call that "**failures**" happened. These are related to skilled behaviors and we can call them "slips and lapses" as well.

Error Types





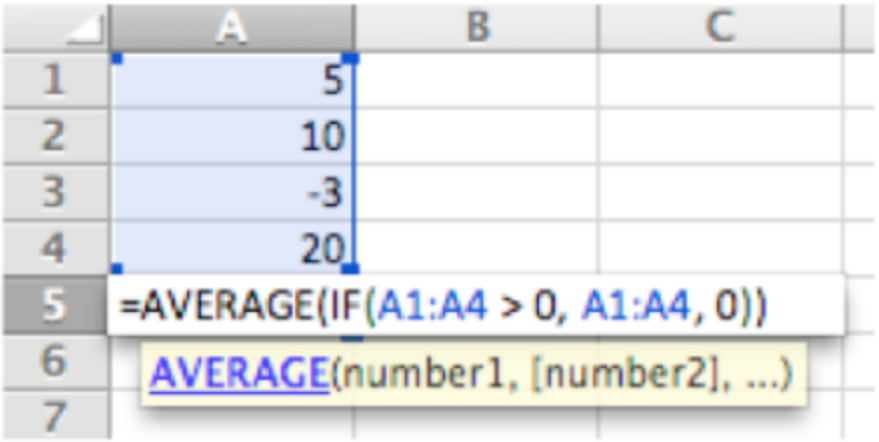
Source: (Medical Device Safety) IEC 62366:2007-10 and ANSI/AAMI HE75:2009

Here's a Venn diagram that shows the classification into mistakes and slips/lapses, with a finer categorization of slips discussed below.

Furthermore, we can classify error types based on whether the action was intended or unintended. Slips and lapses are caused by "unintended use" whereas mistakes are owing to the user's "intended use" of inappropriate rules/knowledge to deal with the tasks at hand. Note that a user's action may deviate from the "intended usage" (what the designer specified!). This is called "appropriation." From the human error perspective, this can be considered as "abnormal use" if less proper usage causes errors.

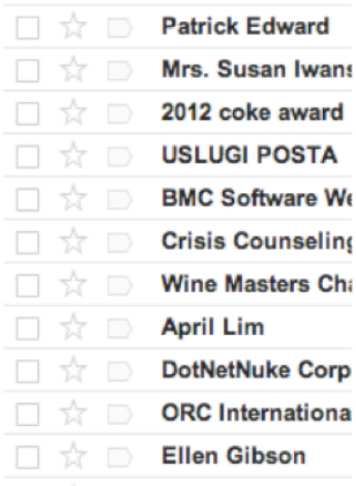
Capture Errors

- A frequent sequence of actions *captures* a less frequent sequence of actions
- Leave your house and find yourself walking to school instead of where you meant to go
- vi :w command (to save the file) vs. :wq command (to save and quit)
- Excel array formulas must be entered with Ctrl-Shift-Enter, not just Enter



Here are some examples of common slips. A **capture slip** occurs when a person starts executing one sequence of actions, but then veers off into another (usually more familiar) sequence that happened to start the same way. A good mental picture for this is that you've developed a mental groove from executing the same sequence of actions repeatedly, and this groove tends to capture other sequences that start the same way. In the text editor vi, it's common to quit the program by issuing the command ":wq", which saves the file (w) and quits (q). If a user intends just to save the file (:w) but accidentally quits as well (:wq), then they've committed a capture error. Microsoft Excel has a curious (and very useful!) class of formulas called array formulas, but in order to get Excel to treat your formula as an array formula, you have to press Ctrl-Shift-Enter after you type it - every time you edit it. Why is this prone to capture slips? Because virtually every other edit you do is terminated by Enter, so you're very likely to fall into that pattern automatically when you edit an array formula.

Description Errors



Kendall Station
Kenmore Station

6.813/6.831

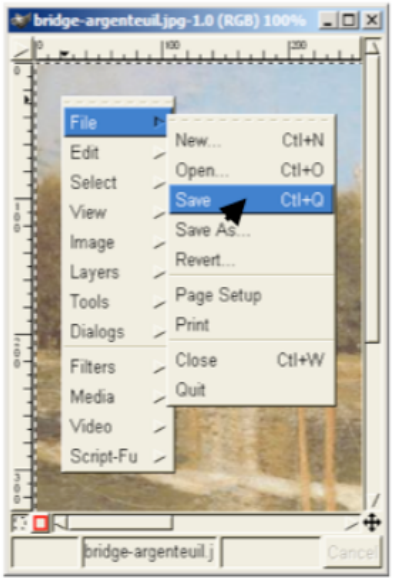
- Actions have similar descriptions
- Description can be visual, spatial, or semantic
- Consistency (same) is good for learning
- Inadvertent similarity (close but not quite) is bad for safety

A **description slip** occurs when two actions are very similar. The user intends to do one action, but accidentally substitutes the other. A classic example of a description error is reaching into the refrigerator for a carton of milk, but instead picking up a carton of orange juice and pouring it into your cereal. The actions for pouring milk in cereal and pouring juice in a glass are nearly identical - open fridge, pick up half-gallon carton, open it, pour- but the user's mental description of the action to execute has substituted the orange juice for the milk.

Some other pairs that may be prone to description slips are shown above. In the GMail interface on the far right, the three icons - checkbox, star, and important - look very similar to each other when unchecked, opening the way to description slips.

Mode Errors

- Modes: states in which actions have different meanings
  - Vi's insert mode vs. command mode
  - Caps Lock
  - Drawing Palette



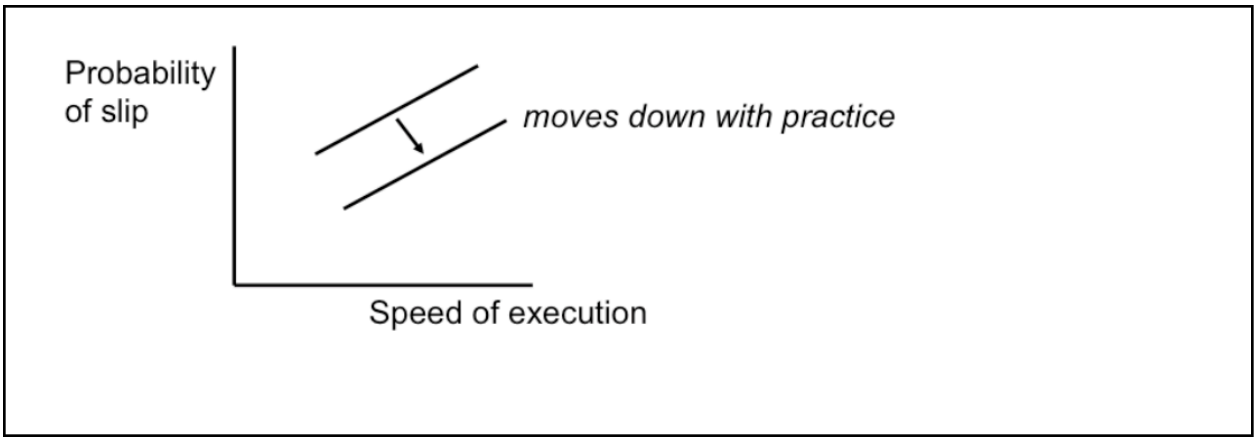
Another kind of error, clearly due to user interface, is a mode error. **Modes** are states in which the same action has different meanings. For example, when Caps Lock mode is enabled on a keyboard, the letter keys produce uppercase letters. The text editor vi is famous for its modes: in insert mode, letter keys are inserted into your text file, while in command mode (the default), the letter keys invoke editing commands.

Mode errors occur when the user tries to invoke an action that doesn't have the desired effect in the current mode. For example, if the user means to type lowercase letters but doesn't notice that Caps Lock is enabled, then a mode error occurs.

Mode errors are generally slips, an error in the execution of a learned procedure, caused by failing to correctly evaluate the state of the interface.

Causes of Slips

- "Strong-but-wrong" effect
  - Similarity
  - High frequency
- Inattention or inappropriate attention
- Speed / accuracy tradeoff



The slips and lapses we've discussed have a few features in common. First, the root cause of these errors is often **inattention**. Since slips and lapses occur in skilled behavior, execution of already well-learned procedures, they are generally associated with insufficient attention to the execution of the procedure, or omission or distraction of attention at a key moment.

Second, the particular erroneous behavior chosen is often selected because of its high similarity to the correct behavior (as in capture and description slips), or of its high frequency relative to the correct behavior (as in capture slips). Very common, or very similar, patterns are strongly available for retrieval from human memory. Errors are often **strong-but-wrong** behavior.

Finally, we can tune our performance to various points on a **speed-accuracy** tradeoff curve. We can force ourselves to make decisions faster (shorter reaction time) at the cost of making some of those decisions wrong.

Conversely, we can slow down, take a longer time for each decision and improve accuracy. It turns out that for skill-based decision making, reaction time varies linearly with the log of odds of correctness; i.e., a constant increase in reaction time can double the odds of a correct decision.

The speed-accuracy curve isn't fixed; it can be moved down and to the right by practicing the task. Also, people have different curves for different tasks; a pro tennis player will have a high curve for tennis but a low one for surgery.

One consequence of this idea is that **efficiency** can be traded off against **safety**. Most users will seek a speed that keeps slips to a low level, but doesn't completely eliminate them.

Error Prevention

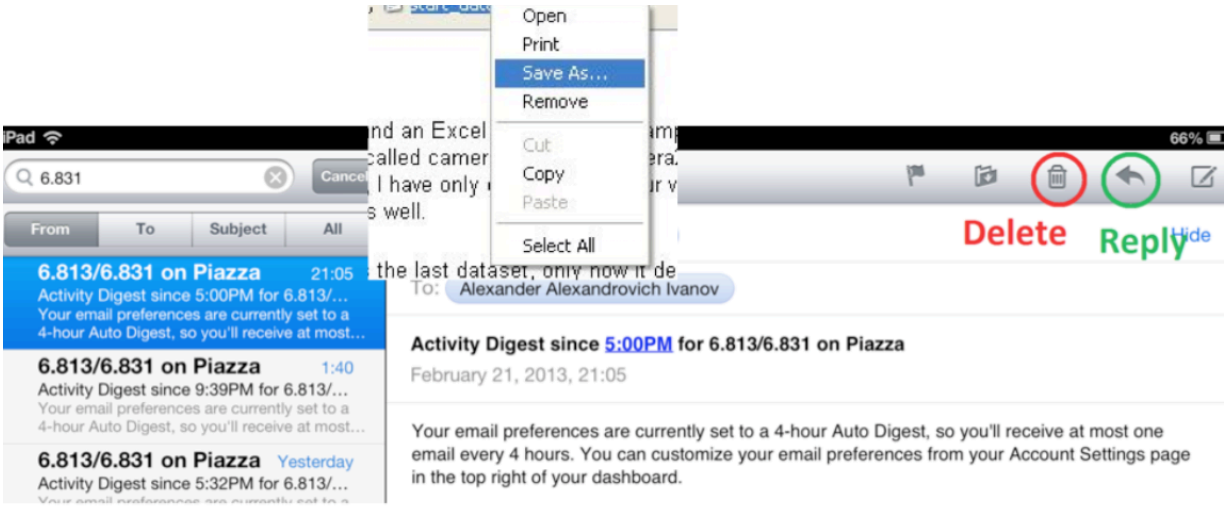
Safety from Capture Errors

- Avoid habitual action sequences with identical prefixes

Let's discuss how to prevent errors of these sorts. In a computer interface, you can deal with capture errors by avoiding very common action sequences that have identical prefixes.

Safety from Description Errors

- Avoid actions with very similar descriptions
- Keep dangerous commands away from common ones



Description errors can be fought off by applying the converse of the Consistency heuristic: different things should look and act different, so that it will be harder to make

description errors between them. Avoid actions with very similar descriptions, like long rows of identical buttons.

You can also reduce description errors by making sure that dangerous functions (hard to recover from if invoked accidentally) are well-separated from frequently-used commands. Outlook 2003 makes this mistake: when you right-click on an email attachment, you get a menu that mixes common commands (Open, Save As) with less common and less recoverable ones - if you print that big file by mistake, you can't get the paper back. And if you Remove the attachment, it's even worse - undo won't bring it back! (Thanks to Amir Karger for this example.)

Unfortunately the Mail app on the iPad **also** makes this mistake - the Delete and Reply buttons are right next to each other. (thanks to Alexander Ivanov for this example)

Safety from Mode Errors

- Eliminate modes
- Increase visibility of mode
- Spring-loaded or temporary modes
- Disjoint action sets in different modes

There are many ways to avoid or mitigate mode errors. Eliminating the modes entirely is best, although not always possible. Modes do have some uses - they make command sets smaller, for example. When modes are necessary, it's essential to make the mode visible. But visibility is a much harder problem for mode status than it is for affordances. When mode errors occur, the user isn't actively looking for the mode, like they might actively look for a control. As a result, mode status indicators must be visible in the user's locus of attention. That's why the Caps Lock light, which displays the status of the Caps Lock mode on a keyboard, doesn't really work.

Other solutions are spring-loaded or temporary modes. With a spring-loaded mode, the user has to do something active to stay in the alternate mode, essentially eliminating the chance that they'll forget what mode they're in. The Shift key is a spring-loaded version of the uppercase mode. Drag-and-drop is another springloaded mode; you're only dragging as long as you hold down the mouse button. Temporary modes are similarly short-term. For example, in many graphics programs, when you select a drawing object like a rectangle or line from the palette, that drawing mode is active only

for one mouse gesture. Once you've drawn one rectangle, the mode automatically reverts to ordinary pointer selection.

Finally, you can also mitigate the effects of mode errors by designing action sets so that no two modes share any actions. Mode errors may still occur, when the user invokes an action in the wrong mode, but the action can simply be ignored rather than triggering any undesired effect.

Safety by Forcing User Interaction

- Interlock: creating mutual dependency in a sequence of actions
- Lockin: keeping specific operations active to avoid accidental stopping
- Lockout: asking users to perform some action to proceed with a task

There are a few ways of preventing human errors: interlock, lockin, and lockout. These mechanisms are widely used in many safety critical devices such as medical tools and automobiles.

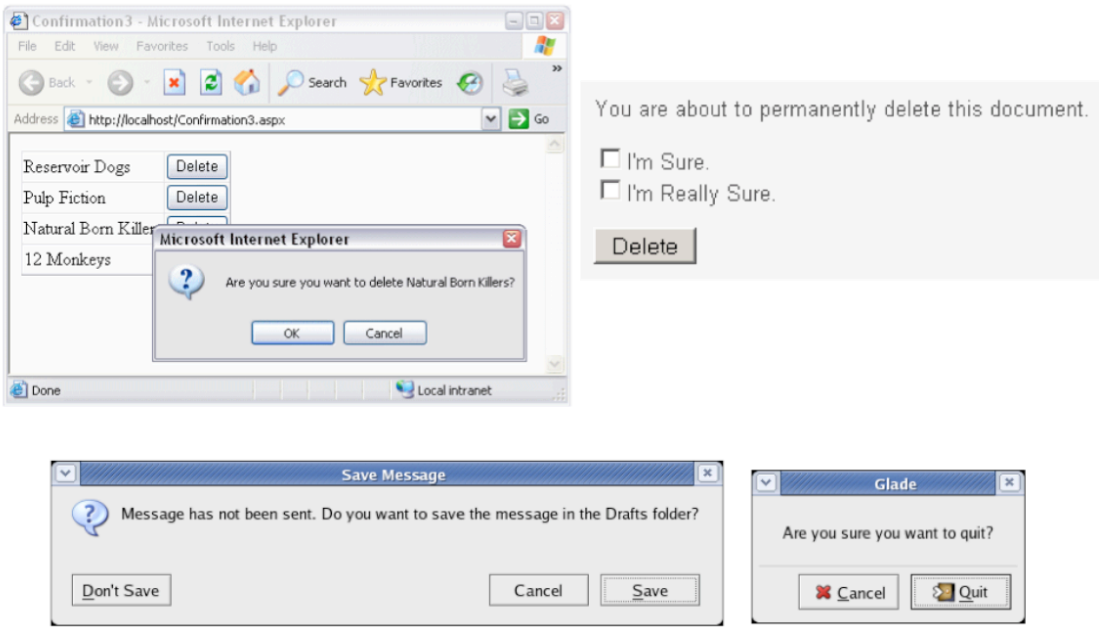
Interlock is to force user interactions (involving changing a machine's state) to happen in a specific sequence, thereby creating a mutual dependency between two interactions. It is typically used to prevent a machine from harming its operators or damaging itself. Several examples in our lives include the following: you cannot open the door in a moving elevator, when you open the door of the microwave, its operation is stopped, your car can be started only when it is in the park mode, and an ATM asks you to remove your card before you take money. There is an ISO standard on interlock: ISO 14119:2013(en) (Safety of machinery — Interlocking devices associated with guards — Principles for design and selection).

Lockin is similar to the mode, but its goal is to keep a specific operation active, by avoiding accidental stopping. For example, lockin is quite common when you make firmware updates.

Lockout is to require users to perform an unlock interaction to avoid accidental access; e.g., a stopping bar in the stair. In software user interface design, we can implement lockout in several different ways. A simple example is to prevent a user from proceeding with a task for some time (known as task lockout). A task lockout can be inserted in different points of time (e.g., when starting a task, or transitioning from one sub-task to another sub-task). For example, when a user launches an app, it may present a lockout screen, by asking users to carefully read a safety procedure before proceeding. In a

number entry task, we can temporarily lockout the task by simply asking a user to check the number that he/she just entered. According to a recent study, this simple “task lockout” of artificially inserting a small amount of waiting time for checking during the number entry task really helps improve the accuracy. Of course, if the lockout duration is too long, users are likely to do other tasks or switch tasks (see Now Check Your Input: Brief Task Lockouts Encourage Checking, Longer Lockouts Encourage Task Switching, CHI 2016). There is an interesting concept called design frictions for mindful interactions (by Cox et al., Design Frictions for Mindful Interactions: The Case for Microboundaries, CHI 2016). This kind of design friction creates “gulf of execution” in interactions, thereby making people to be mindful when interacting with devices.

Confirmation Dialogs



An unfortunately common strategy for error prevention is the **confirmation dialog**, or “Are you sure?” dialog.

It’s not a good approach, and should be used only sparingly, for several reasons:

- Confirmation dialogs can substantially reduce the efficiency of the interface. In the example above, a confirmation dialog pops up whenever the user deletes

something, forcing the user to make two button presses for every delete, instead of just one. Frequent commands should avoid confirmations.
- If a confirmation dialog is frequently seen - for example, every time the Delete button is pressed - then the expert users will learn to expect it, and will start to include it in their habitual procedure. In other words, to delete something, the user will learn to push Delete and then OK, without reading or even thinking about the confirmation dialog! The dialog has then completely lost its effectiveness, serving only to slow down the interface without actually preventing any errors.

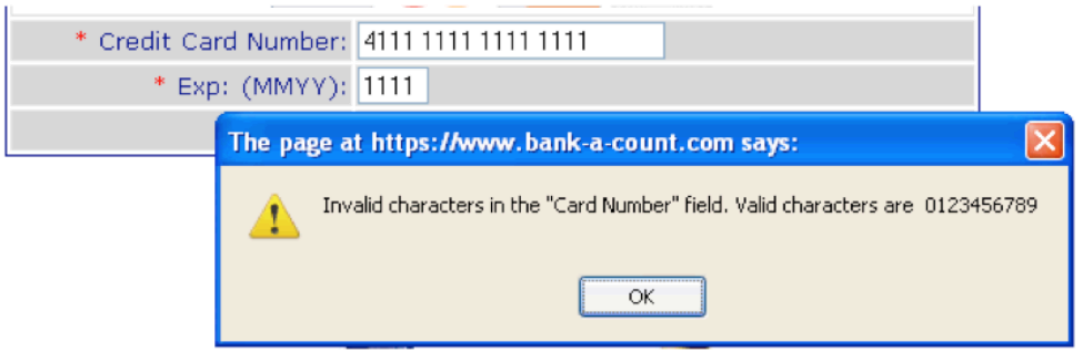
In general, reversibility (i.e., **undo**) is a far better solution than confirmation.

Even a web interface can provide at least single-level undo (undoing the last operation). Operations that are very hard to reverse may deserve confirmation, however. For example, quitting an application with unsaved work is hard to undo - but a well-designed application could make even this undoable, using automatic save or keeping unsaved drafts in a special directory.

Error Messages

Writing Error Message Dialogs

- Best error message is *none at all*
    - Errors should be prevented
    - Be more flexible and tolerant
    - Nonsense entries can often be ignored without harm



Source: "No Dashes Or Spaces" Hall of Shame

Finally, let's talk about how to write error messages. But before you try to write an error message, stop and ask yourself whether it's really necessary. An error message is evidence of a limitation or lack of flexibility on the part of the system - a failure to prevent an error or absorb it without complaint. So try to **eliminate the error** first.

Some errors simply aren't worth a message. For example, suppose the user types "abc" into the font size combo box. Don't pop up a message complaining about an "invalid entry". Just ignore it and immediately replace it with the current font size. (Why is this enough feedback, for a font size combo box?) Similarly, if the user drags a scrollbar thumb too far, the scrollbar doesn't pop up an error message ("Too far! Too far!"). It simply stops. If the effect of the erroneous action is easily visible, as in these cases, then you don't have to beat the user over the head with a superfluous error message.

The figure shows an example of an error message that simply shouldn't happen. Forbidding dashes and spaces in a number that the user must type, like an account number or credit card number, is poisonous to usability. (Why are dashes and spaces helpful for human perception and memory?) There's a great collection of error messages like this at the No Dashes or Spaces Hall of Shame (http://www.unixwiz.net/ndos-shame.html).

Be Precise and Comprehensible

- Be precise
- Restate user's input
- Speak the user's language

Assuming you can't design the error message out of the system, here are some guidelines for writing good ones.

Be precise. Don't lump together multiple error conditions into a single all-purpose message. Find out what's really wrong, and display a targeted message.

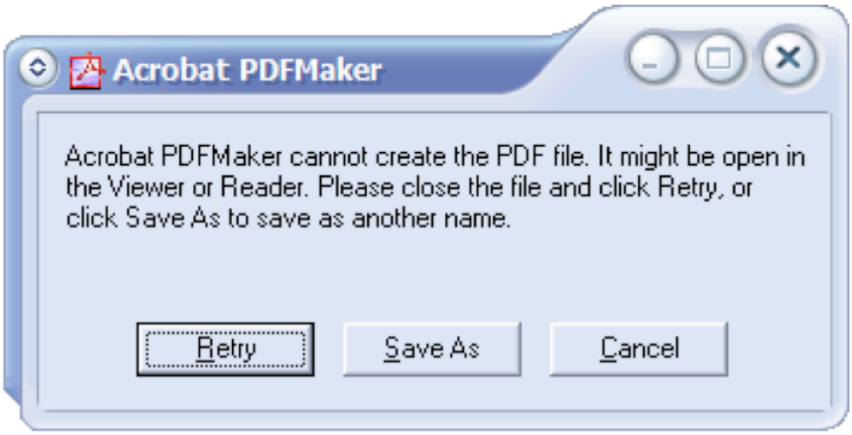
If the error is due to limitations of your system, like sizes or allowed characters, then be specific about what the limitations are, so that the user can adapt. (Then ask yourself why you have those limitations!)

It often helps to restate the user's input, so that they can relate what they did to the error message, and perhaps even detect the problem immediately ("oh, I didn't mean paper.doc…")

In error messages, it's particularly important to speak the user's language, and avoid letting technical terms or details like exceptions and stack traces leak through.

Suggest Reasons and Solutions

- Give **constructive** help: why the error occurred and how to fix it

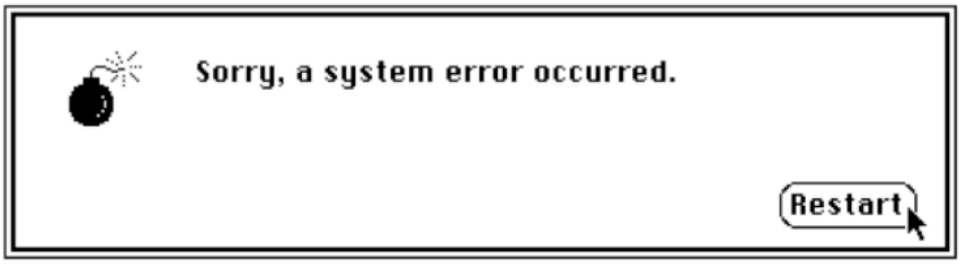


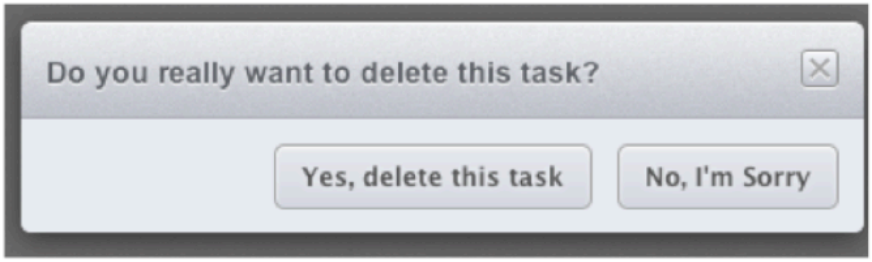
Next, your message should be **constructive**, not just reporting the error but helping the user correct it. Suggest possible reasons for the error and offer ways to correct

them–ideally in the error message dialog itself. Here's a good example from Adobe Acrobat.

Be Polite

- Be polite and nonblaming



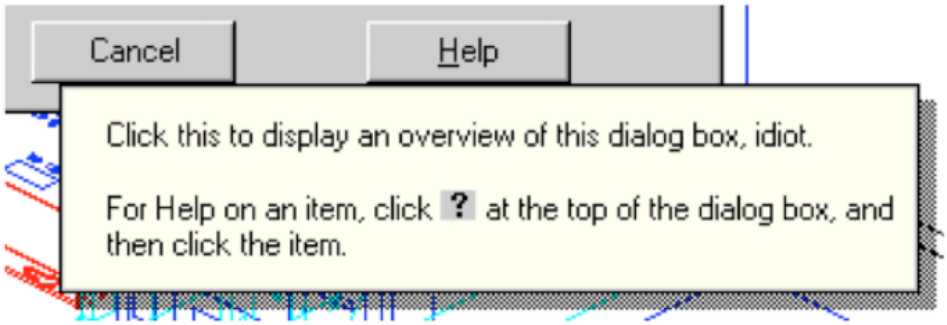


Finally, be polite. The message should be worded to take as much blame as possible away from the user and heap the blame instead on the system. Save the user's face; don't worry about the computer's. The computer doesn't feel it, and in many cases it is the interface's fault anyway for not finding a way to prevent the error in the first place. It's interesting to contrast what the original 1984 Mac said when it crashed (an apology!).

The confirmation dialog on the bottom isn't an error message, strictly speaking, but it does show incorrect attribution of blame. The user shouldn't have to apologize!

Avoid Loaded Words

- Fatal, illegal, aborted, terminated



Many words that are unfortunately common in technical error messages have emotionally-charged meanings in ordinary language; examples include “fatal”, “illegal”, “abort”, etc. Avoid them. Use neutral language. Windows and DOS have historically been littered with messages like these.

The tooltip shown at the bottom isn’t strictly an error message, but it actually appeared in a production version of AutoCad! As the story goes, it was inserted by a programmer as a joke, but somehow never removed before release. Even as a joke, it demonstrates a lack of respect for the intelligence of the human being on the other side of the screen. That attitude is exactly wrong for user interface design.