Open access • Proceedings Article • DOI:10.1145/1124772.1124832

Answering why and why not questions in user interfaces — Source link

Brad A. Myers, David A. Weitzman, Amy J. Ko, Duen Horng Chau

Institutions: Carnegie Mellon University

Published on: 22 Apr 2006 - Human Factors in Computing Systems

Topics: User modeling, User interface, User story, User interface design and Natural user interface

Related papers:

- Designing the whyline: a debugging interface for asking questions about program behavior
- Why and why not explanations improve the intelligibility of context-aware intelligent systems
- Assessing demand for intelligibility in context-aware applications
- Coupling a UI framework with automatic generation of context-sensitive animated help
- Intelligibility and accountability: human considerations in context-aware systems

Share this paper:

View more about this paper here: https://typeset.io/papers/answering-why-and-why-not-questions-in-user-interfaces-23ururdqmz

Answering Why and Why Not Questions in User Interfaces

Brad Myers, David A. Weitzman, Andrew J. Ko, and Duen Horng Chau
Human Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213
bam@cs.cmu.edu
http://www.cs.cmu.edu/~bam

ABSTRACT
Modern applications such as Microsoft Word have many automatic features and hidden dependencies that are frequently helpful but can be mysterious to both novice and expert users. The "Crystal" application framework provides an architecture and interaction techniques that allow programmers to create applications that let the user ask a wide variety of questions about *why* things did and did not happen, and how to use the related features of the application without using natural language. A user can point to an object or a blank space and get a popup list of questions about it, or the user can ask about recent actions from a temporal list. Parts of a text editor were implemented to show that these techniques are feasible, and a user test suggests that they are helpful and well-liked.

Author Keywords
Why, Help, Questions, Natural Programming.

ACM Classification Keywords
D.2.2 Design Tools and Techniques: *User interfaces*; D.2.6 Programming Environments: *Graphical environments*; H.5.2 User Interfaces: *Interaction styles, Training, help, and documentation*; D.2.11 Software Architectures.

INTRODUCTION
One of the classic guidelines for user interface design is to have "visibility of system status" to "keep users informed about what is going on" [18]. And yet, in an informal survey of novice and expert computer users, everyone was able to remember situations in which their computer did something that seemed mysterious. For example, sometimes Microsoft Word automatically changes "teh" into "the", but it does not change "nto" into "not". The spacing above a paragraph can be affected by properties in the "Format Paragraph" dialog box, along with the heights of the actual characters on the first line of the paragraph (even the heights of invisible characters such as spaces). In the Windows desktop and Windows Explorer "Icon" view, sometimes the icons go where you put them but sometimes they auto-arrange into columns. A command that hides all the windows can be invoked by accident, making users wonder where their windows went.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CHI 2006, April 22–27, 2006, Montréal, Québec, Canada.
Copyright 2006 ACM 1-59593-178-3/06/0004...$5.00.

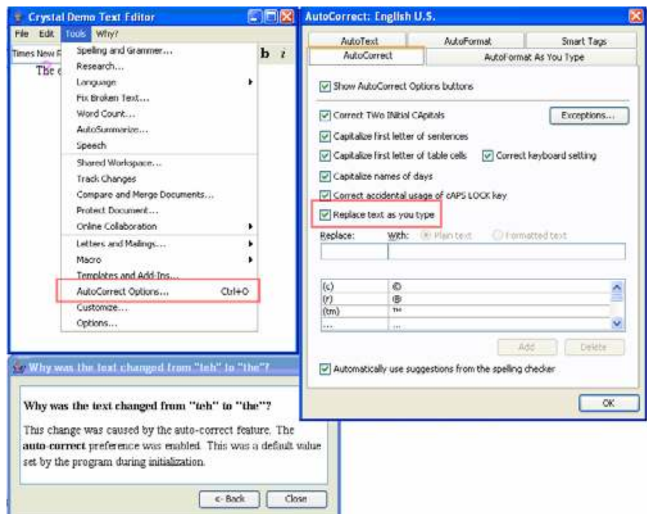

Figure 1: The answer for why "Teh" was changed into "The". The pink "?" in the upper left shows where the F1 key was hit.

All of these features, and the dozens of others that we collected (and that the reader can undoubtedly think of), are quite useful to most users, and have been added to user interfaces because they help most people most of the time. However, when a novice or expert is unfamiliar with these features, or when something happens that is *not* desired, there is no mechanism to figure out *why* the actions happened, or how to control or prevent them. It is even more difficult when an expected action does *not* happen, for example, why did the spelling *not* get corrected? No help system built into any of today's systems can answer these questions. As applications inevitably get more sophisticated, such a facility will be even more necessary.

Inspired by the Whyline research [11] that answers "why" and "why not" questions about a program's execution to aid debugging, we created an application framework called Crystal that helps programmers build applications that can answer questions about an application (see Figure 1). Crystal provides **C**larifications **R**egarding **Y**our **S**oftware using a **T**oolkit, **A**rchitecture and **L**anguage. The idea is that the system makes things "crystal clear." At this point Crystal is

primarily a feasibility demonstration, but it does show that a system can helpfully answer users' "why" and "why not" questions.

Instead of supporting natural language, Crystal builds question menus dynamically based on the current state of the application. The user can ask questions either by hitting a key (currently F1) while the mouse cursor is over the item of interest, as was done in Figure 1, in which case Crystal automatically builds a menu of questions about the objects under the mouse. Crystal provides invisible objects under every point in the window so users can ask questions by pointing to where there are apparently no objects, such the white space around paragraphs.

Alternatively, a "why" menu displays a list of the last few operations that were or were not performed. This includes explicit user actions (e.g., "hitting the 'Backspace' key"), along with automatic actions like spelling correction, and other actions which are normally not logged (e.g., hiding windows). This list also includes actions that the user tried to perform but did not actually execute, such as hitting Control-C for Copy with nothing selected. The application designer can add to the menus questions about other things that did *not* happen which might be mysterious to users. Examples include when interdependencies or constraints prevent an object from moving or cause automatic correction to not happen.

In response to any of these questions, Crystal displays a window containing an automatically-created explanation (see the bottom-left of Figure 1). Whenever possible, the elements of the user interface that contributed to the value are displayed, and a red highlight is put around the user interface controls (also called "widgets") relevant to the question. In Figure 1, the "Replace text as you type" checkbox of the AutoCorrect dialog is highlighted. In cases where the user interface controls cannot be so easily displayed, Crystals adds a "How Can I…" question to the bottom of the explanation window, to allow the user to ask how to control the features that were involved in the operation. Other systems have supported such "How Can I" questions, but not in the context of "why" questions, and Crystal also differs in that it automatically determines how to enable the actions.

Like the Whyline [11], Crystal must store extra information about a program's execution to support answering the questions. Therefore, the question-answering cannot simply be plugged into an existing application like Microsoft Word. Instead, the application must be built in such a way as to collect the appropriate information during execution. The Crystal framework adds novel extensions to the command-object model [16] to store the appropriate information. This makes it easy to build applications which will support the asking of questions.

To demonstrate the effectiveness of this framework, we used it to build parts of a sample text editor which has some automatic transformations like Microsoft Word. We then used this editor in a small user study. The results suggest that Crystal is effective in teaching users about these complex features, and the interaction techniques were easy to use and well-liked. Participants with the "why" features were able to complete about 30% more tasks than those without, and of the tasks completed, participants with the "why" features were about 20% faster.

The Crystal framework is primarily intended to help explain complex behaviors and interdependencies among the various features. It is *not* intended to help the end-user find out why things happened if the programmer introduced bugs into the application. The assumption that Crystal makes is that all the resulting behaviors are intended. If the programmer does not know why something happens, it is unrealistic to expect end-users to!

The rest of this paper summarizes the related work, describes the user interface features, and then explains in detail the software architecture that makes asking the questions possible. The user study is then described, followed by future work and conclusions.

RELATED WORK

Help systems for interactive applications have been studied extensively. Norman discusses two important "gulfs" in people's use of their systems [19]. Many help systems (e.g., [7] [13] [20] [22]) are designed to help with the *gulf of execution*: teaching users how to perform actions, primarily to learn about a command they already know the name of, or learn how to perform tasks. For example, Cartoonist [22] displays animated help showing the steps required, but it must explicitly be given the name of a command or task. In contrast, we believe that Crystal provides the first help system to specifically target the *gulf of evaluation*: helping users interpret what they are seeing on the screen and determine how to fix it if it is not what they intended.

Many recent help systems focus on giving tutorials for how to use a system. For example SmartAidè [20] uses AI planning methods to give step-by-step instructions when the user has a goal in mind but does not know how to execute it. "Stencils" focuses the user's attention on certain parts of the interface during a tutorial to prevent errors [8]. The Crystal framework would probably be helpful in building such systems, since it provides an explicit representation between the user actions and the underlying behaviors, but creating tutorials using Crystal is left for future work.

AI-based question answering systems (e.g., [13] [23]) focus on improving the effectiveness of queries that use natural language, which Crystal avoids by generating popup menus containing specific relevant questions .

A number of systems have allowed the user to go into a special mode and click on controls in the interface to get help on them. This was available, for example, in the first version of LabView [17] in 1986, and the "?" icon works this way in some Windows dialog boxes. Eclipse will display "*infopops*" when the user presses F1 over any user

interface widget [7]. The infopops can contain links to various topics. In these kinds of systems, however, the help text is statically written by the programmer and does not help with questions about why actions did or did not happen. In Crystal, the question and answer text is automatically generated from the program's execution history.

In its answers, Crystal highlights the actual widgets of the interface. This approach has been used in Apple Guide [10], Stencils [8], and the "Show me" feature of some modern help systems. A difference from these is that Crystal automatically determines which widgets should be highlighted.

The only systems we are aware of that try to use tracing and dependency information to help users are programming systems such as spreadsheets and debuggers. For example, Microsoft Excel 2003 will show the cells on which the current cell depends. Forms/3 goes further in providing visualizations that try to focus the user's attention on from where faulty values may have come [21]. Production systems, such as ACT-R, have long had the ability to ask why productions did or did not fire [3], and the Whyline [11] generalizes this to any output statement in the program. Dourish [5] speculates about how an open data model [9] [14] might help applications explain their behavior, and provides motivation and technical guidelines, but does not describe any implementation. We are not aware of any applications for end users that dynamically generate a list of "why" questions about the context, or dynamically create the answers based on the history of users' actions.

USER INTERFACE

Crystal makes contributions in two areas: the interaction designs for asking and answering questions, and the framework to make implementing this easier. Research has shown that users are often reticent to use help systems and that the help system's own user interface can be a barrier to its use [6]. Therefore, a key requirement for Crystal is that it be very easy to invoke and that the answers be immediately helpful.

To address these issues, we designed the interface to the "why" system with just two simple interaction techniques: the F1 key and the "why" menu. The "why" menu also contains an item to go into a mode that allows invoking location-based questions, in case the user does not know how to use the F1 key. Our observations suggest that virtually all of the user's questions will be about things that are visible (or invisible in the case of white space) in their application, or things that happened recently.

Which Questions To Include

The next important design issue is what questions belong in the menus. In a simple direct manipulation application, such as a drawing editor, the only things that happen are what the user explicitly does, so the question menu will simply have one entry for each user action. In this case, the question menu is automatically built by Crystal from the commands that are executed. However, any sophisticated application will also have situations where there are hidden states or invisible dependencies that affect what users see. Examples include when a setting in one part of the user interface controls whether other things happen, such as the auto-corrections in Figure 1, and whether meta-information, such as paragraph marks (¶), are displayed or not. These must be added to the question menus as well. However, the application designer must guard against having too many questions in the menu, because then it will take too long for the user to find the desired question. Crystal therefore provides a way for the designers to note that certain actions should be omitted from the question menus.

For example, when implementing the sample text editor, we decided not to add regular typing to the menu, because it seemed unnecessary to let the user ask why "b" appeared, with the answer being "because you typed it." Similarly, we do not add questions about why characters move around (characters move when you type before them). In general, these are excluded because the actions and their feedback are so common and so immediate that users already know the answers. In other application domains, there are similar types of basic operations that would be excluded by the application designer (such as back and forward in a web browser, automatically marking e-mails as read after five seconds, changing tools in Photoshop, and other actions with immediate and direct visual feedback). Note that designers use similar heuristics today to decide what should go into the undo menus, and at what granularity – scrolling is not on the undo menu at all, and typing is grouped into chunks for undo.

In the sample text editor, there are questions for all other explicit user actions, including when typing causes the selected text to be deleted. If the editor supported complex mechanisms that moved text in non-intuitive ways (such as the widow/orphan control in Word), then these would be added to the menu as well.

The "why" menu also contains some actions that did *not* happen. Of course, an infinite number of different things could be added, but users only need to be able to find out about things they expected to happen. Some of these can be handled automatically by Crystal, including non-actions that stem from explicit user input. For example, Crystal adds to the menu questions for keystrokes that have no effect, such as typing Control-C with nothing selected (see Figure 2). Also added are questions about actions that did not do anything because they were explicitly disabled. For example, if the auto-correct shown in Figure 1 was turned off, and the user types "Teh", the menu will let the user ask why it was not corrected (see Figure 2). For background tasks, however, the application designer will have to notify Crystal when menu items should be added. The programmer specified that spelling corrections should be added to the menus, but "Why Not" questions are not added for words that are spelled correctly and therefore not corrected, since this would quickly fill up the menu with questions that are never likely to be of interest.

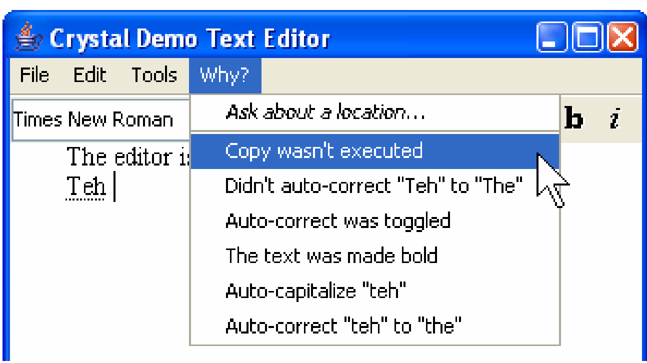

Figure 2: "Why" menu. The top item lets the user click for where to get help. The next two actions in the menu did *not* happen.

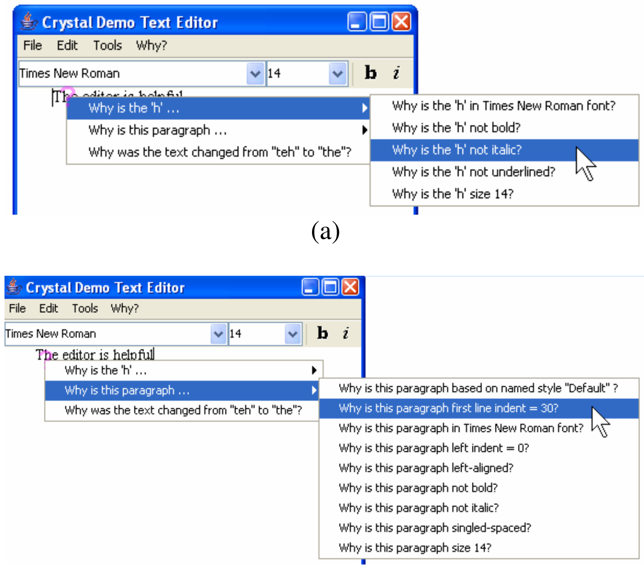

Figure 3: Menus resulting from hitting F1, showing sub-menus for the character (a) and paragraph (b) properties.

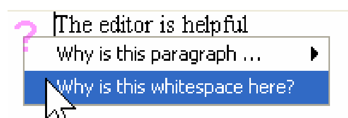

Figure 4: Questions about blank areas when hit F1.

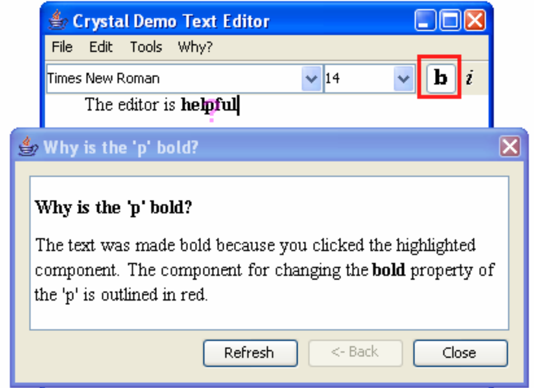

Figure 5: The answer to "Why is the 'p' bold?", when it was because the user set the property using the toolbar button.

Designing the Menus

When the F1 key is hit, Crystal looks at all objects under the cursor to generate the list of questions. For example, before getting the windows shown in Figure 1, the menu at the left of Figure 3 would have appeared. The first level menu has questions about the character and paragraph under the mouse, and any global operations performed on that object. Figure 1 resulted from choosing the last item in the first menu. In Figure 3-a, the user has selected the question about the properties of the character "h".

The questions in the menus are designed to feature the values in an easy-to-find place (at the end of each question) so that a quick scan will show all the properties' values. To display each value, Crystal uses a variety of built-in rules so the menus are concise yet readable. For Boolean properties, the value name or "not" the value name is used, such as "bold" or "not italic". For numeric properties, we use *property = value*. These automatic rules can be augmented by the designer with rules for application-specific types. For example, for the sample text editor, we added a custom rule to just use the style name for style values (such as "Default" in Figure 3).

If the F1 key is hit while the mouse is over a blank part of the window, Crystal includes questions in the menu about why that white space is there. In Figure 4, the paragraph is listed because it has an invisible portion that extends to the left edge of the window, since paragraphs control indenting. The designer of the editor has also added to the menu an additional question about whitespace, which summarizes all the different contributions to that whitespace (since character and paragraph properties might both be involved in other situations).

Like Eclipse [7], hitting F1 while the mouse cursor is over a control, such as a dialog box or a menu item, will provide help for that control. If the item is grayed out, Crystal will generate an explanation for why it is disabled.

Providing Useful Answers

Answers to the questions typically have two parts: a textual explanation and highlighting of the relevant user interface controls (see Figure 5). The motivation is that users typically want to know more than *why* something happened—they also want to know *what they can do about it*, such as changing it to be different. Therefore, whenever possible, answers highlight specific actions that users can take.

When the referenced control is in a dialog box, Crystal also highlights all the controls necessary to making it appear, so the user does not have to figure out how to get what the answer discusses to happen. For example, in Figure 1, Crystal has highlighted the AutoCorrect Options menu item in the Tools menu, and the specific control used on the resulting dialog. All dialogs are "live" while they are highlighted, so the user can operate them normally. This will often save the user a number of steps if the property needs to be changed. In fact, it is sometimes quicker to use the F1 fea-

ture to get to the desired dialog box instead of navigating to it, even when the user knows why things happened.

While we expect that the controls and dialog boxes of the application will be the primary focus for the user's answers, the textual explanation is necessary in some situations, such as when there is a chain of causes for the situation. For example, Figure 6 shows the answer explaining why the text is size 20, which is inherited from its style. The explanation is also useful when the user wants to learn how the application works in detail.

When there are multiple causes and actions as part of the explanation, Crystal adds to the bottom of the answer window a link for each one (see Figure 6). When clicked, the text window provides the answer and the appropriate controls are highlighted. The back button in the answer window can then be used to return to the original question. When the user closes the answer window, the highlighting is removed from all controls.

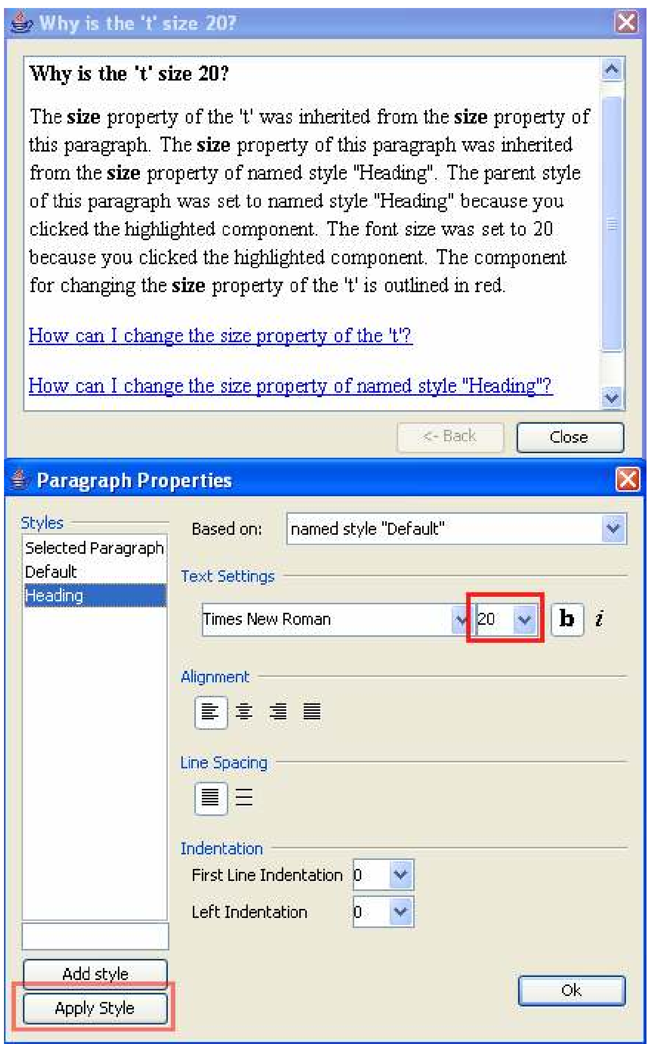

Figure 6: The answer shown for when a property's value is inherited from a style.

APPLICATION FRAMEWORK

An important contribution of this research is an object-oriented framework that makes it easy to create applications that support "why" and "why not" questions. The Crystal framework is implemented on top of Java Swing 1.5 and makes use of the standard Swing controls and architecture. The key additions in the Crystal framework are abstractions to represent **application objects** and their **properties**, and **command objects** that support undo and questions. The result is a framework where only a small amount of additional code is needed to support the "why" questions, beyond what is needed anyway to support undo. We used this framework to implement a sample text editor as a test application. We chose a text editor because it is a particularly difficult kind of application to build. Also, the Microsoft Word text editor contains many complex features that we wanted to see if our system could help explain. Implementing a graphical editor, as has been used to test most previous frameworks [2, 16], would be straightforward.

Hierarchical Command Objects

Crystal uses a "Command Object model" [2, 16] to implement all of the actions. As commands are executed, they are stored on a command list which serves as a history of all the actions that have been taken. This command list is used for undo and the why menus.

Crystal uses *hierarchical* command objects [16]. The top-level command objects are all the user-executed commands (like when the user clicks on a menu item). The lower-level command objects are for the individual actions that a command may include. For example, setting some text to the "Heading" style might change the size, the font, and make the text bold. Crystal separates these into three different sub-commands of the `Set-Style` top-level command.

Each command object contains a variety of methods and fields, as shown in Figure 7. The first six are typical of other command object systems [2, 16], but the second six are novel with Crystal, and are described next.

`Dependencies`: Crystal needs to know the dependencies among commands and values. In particular, many commands' actions depend on the values of controls. For example, the auto-correct command of Figure 1 depends on the value of the `Replace-Text-As-You-Type` property, and the answer wants to describe this for users. Using the saved old values, the answer generator can fetch the value of the control at the time when the command was executed. This allows Crystal to generate a message like "the auto-correct preference was disabled" even if the property is now enabled. When values are inherited for properties, such as when the font size for a character comes from a named style, the `Dependencies` parameter is used to record where the value came from.

`Invoking-Control`: Each command records the specific control used to invoke this command, since there may be multiple ways to initiate any command (e.g., a keyboard

Name	Function
Do-Method	Performs the action, e.g. changes the font to bold
Undo-Method	Undoes the action
Redo-Method	Redoes the action
Object-Modified	Object affected by this action, so the command can be undone.
Enabled	Boolean to determine if action can be invoked now
Label	String that describes this command
Dependencies	Which properties of which objects are used by this command
Invoking-Control	Which control was used to invoke this command
Questions-Method	Supports application-specific questions
Undoable/Undone	Field that notes whether this command was undone yet
Show-In-Why-Menus	Boolean that flags whether this command should appear in Why menus

Figure 7: Fields and methods of the command objects in Crystal. Properties in bold are novel.

key, a toolbar button and a menu item can invoke the bold command). The `Invoking-Control` value is used to highlight the control in red as part of answers.

`Questions-Method`: When more specific questions and answers are needed for an application, the designer can implement this method. It can also be useful when the designer wants to improve the naturalness of phrasing of the answers. The method returns an object that contains a method to generate the corresponding answer. This is used in the sample text editor for example, by the background auto-correction process. For standard property setting (e.g., "make bold") and actions like creation and deletion, Crystal automatically creates the questions and answers, and the designer does not need to supply a method here.

`Undoable/Undone`: Whether this command can be undone, and if so, whether it was undone yet.

`Show-In-Why-Menus`: As discussed above, the programmer might determine that some commands should not be shown to the user as part of "why" menus even though they are undoable. For example, the Crystal text editor allows regular typing to be undone, but does not add to the "why" menus. The programmer can set `Show-In-Why-Menus` to false for these kinds of commands. Conversely, normally sub-commands are not shown to users in the "why" menus, and instead just the top-level command would be included. However, if the programmer wants to allow the user to ask about a sub-command, then its `Show-In-Why-Menus` can be set to true. An example is that when a new character is typed, the top-level typing command is not displayed in the "why" menus, but if the new character inherits its formatting from a named style, the programmer might want the sub-command that sets the character's properties from the style to appear on the "why" menus, since that may be mysterious to some users.

When a command's `Enabled` property specifies that it is disabled, but the user tries to execute it anyway (e.g., typing Control-C with nothing selected), then a command object is put on the command list with its `Enabled` property set to false to show that it was *not* actually executed. These unexecuted commands allow Crystal to support asking of "why not" questions (Figure 2). Of course, these commands are not undoable, since they were never executed.

Supporting Do/Undo/Redo

In the Crystal framework, an application object, such as a character in a text editor or a rectangle in a graphics editor, is represented as a set of "properties." Examples of properties for a character include the value (e.g., "b") and the font. In order to support undo, the old values of properties must be remembered. The Amulet command objects [16] stored the old values in the command objects themselves. Instead in Crystal (like the Whyline [11]) each property of an object contains a list of all of the old values. Each value is marked with a timestamp, which makes it easy to revert an object to all the correct values at any previous point in the history. If the old values were in the command objects instead, this would require searching all the commands for the appropriate values. Each old value also contains a pointer to the command object that set it, and that command object will contain the same timestamp. Note that making the properties be first-class objects like this is common in many modern toolkits. For example, Swing requires that some properties be objects so that other objects can install property-listeners on them to be notified of changes.

When the user performs undo and then performs a new command, the undone commands can never be redone, so most previous systems throw away the command objects. However, in Crystal, we keep a complete history of all previous actions, even if they were undone, so nothing is ever popped off the command list. Instead, undo causes a new `Undo-Command` object to be added to the head of the list, with a new sub-command that undoes the appropriate action. Then, the command that was undone is marked as undone, so future undo commands will know to skip it.

Note that, as in Microsoft Word, the automatic correction features are added as undoable commands, so, for example, when the user types "teh " and Crystal changes it to "the ", the `auto-correct-command` is added to the command list, so the user can undo the auto-correction separately from undoing the typing of the space.

Connecting Properties of Objects to Commands

As mentioned above, each property of objects in Crystal contains a current value and a list of old values (see Figure 8). Each value is associated with a timestamp and a reference to the command object that caused it to have the current value. Values that are inherited, for example from styles, will still have a local value but there will be an associated property that specifies that the value is inherited. The command object associated with the property will be the

Name	Function
`Value`	Value of the property, e.g., "b"
`My-Object`	Backpointer to the object this is a property of
`Command-Which-Last-Set-Me`	Pointer to the command object which caused the current value to be set
`Inherited`	Tells whether the current value is inherited, and if so, from what other property
`Old-Values`	Time-stamped list of previous values
`Show-In-Why-Menus`	Whether this property should be shown in the "why" menus
`Invoking-Controls`	All the controls that could change this property's value

Figure 8: Fields and methods for Properties of Objects

one that caused the inheritance to happen, and that command object will contain the reference to where the value came from. For example, if a character's font size is 18, which is inherited from a style named "Header", the character's `font-size` property will contain a value 18 with a reference to an `Inherit-From-Style-Command` object, which in turn will reference the `Header` style object. The character will also have an internal `Font-Size-Inherited` property with the value `true`.

Properties in Crystal have a number of additional parameters beyond those needed just to support undo (see Figure 8). Internal properties like `Font-Size-Inherited` have `Show-In-Why-Menus` set to `false` so they will not be made visible to users in the "why" menus. Each property also knows the full set of controls that can affect it directly. For example, the bold property of a character knows about Control-B, the "Toggle Bold" item in the Edit menu, and the "b" button in the toolbar. However, the character bold property does *not* need to know about the "b" button in the paragraph window, since that operates on the bold property of paragraph styles, and when appropriate, Crystal can deduce that it was used by following the dependency information. The list of controls is used to tell the user how the property can be changed.

To explain to the user how values were derived for properties that are never explicitly set, the application designer must add a special non-undoable command to the beginning of the command list which represents all the default, initial values. Then, question and answers can be handled automatically by Crystal, as can be seen in Figure 1, where `auto-correct` has its default value. For systems such as Microsoft Word where initial values can come from many different places: such as various options, `Normal.dot`, etc., the designer would add multiple initialization command objects with custom question methods, so each can describe how the user would change the corresponding default value.

Generating the List of Questions

Generating the list of questions for the "why" menu is straightforward. It is just the last few user-visible items in the command list. Note that it will often include more than what would be available in the undo history, since unexecuted commands and the undo commands themselves will be in the "why" menu. As discussed above, some commands, such as regular typing, are not added to the "why" menu as controlled by the `Show-In-Why-Menus` flag on the commands. The questions used both fixed and dynamic information about what happened (as shown in Figure 2).

Generating the list of questions for the F1 menu is more involved. First, Crystal uses a Swing mechanism to iterate through the components under the mouse, and checks each to see if it implements the `Question-Askable` interface, and if so, calls it. There are three basic ways this interface is implemented by the Crystal framework.

The first is used when F1 is hit on a Swing control, such as a menu or toolbar item, and then associated the command object is used. The programmer can provide a string explaining what the command does. The `Enabled` property of the associated command is automatically checked to see if an additional question should be generated about why the control is disabled. In this case, the programmer can provide a string to explain how to make it be enabled.

The second way is used for objects that have properties. In this case, the framework can handle the questions and answers without help from the programmer. All the user-visible properties of the object, along with their current values, are added in a sub-menu, as shown in Figure 3.

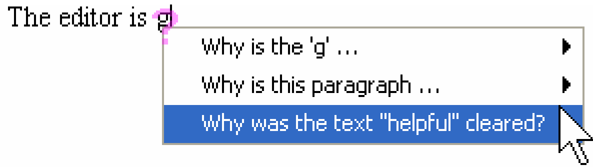

Figure 9: The user typed "g" in Figure 5 while "helpful" was selected, so it was deleted. Crystal inserts an invisible marker in the text so a question will appear about the deleted object.

The third way is used for describing why graphical objects were created or deleted, and is also automatically handled by Crystal. All graphical objects have a pointer to the command that created them so it can be added. Auto-correction is actually implemented as a special kind of create, so a question about auto-correction will be displayed for the appropriate text. However, in this case we added a custom question method to specifically describe the automatic features and dependencies. Objects that are deleted by the user leave invisible objects where they used to be, linked to the commands that deleted them. In a regular graphical editor, this would make it easy to ask about the object that used to be at a location. In the sample text editor, the objects are invisible markers that flow with the text (see Figure 9).

In the text editor, we added a custom method for whitespace that adds an extra question that asks about the whitespace itself. Alternatively, the programmer can provide special invisible objects in all the blank areas, and let them generate questions about why the area is empty.

Generating the Answers

For most questions, Crystal has built-in techniques for generating the answers. For properties of objects (e.g., "Why is the 'g' not bold"), the answer for why it has its current value is provided by showing the operation that *caused* it to have that value, and recursively, why *that* operation happened. Therefore, asking about a property of an object is the same as asking about the command that caused that property to have its current value. This observation was also made by the Whyline study [11] where the "Why is…" questions that were originally in the menus were removed because users were confused about the difference from the "Why did" questions.

For a property that was set locally on the object (such as a character that was explicitly set to bold), the answer says that it was set by the user, as in Figure 5. The corresponding control is also highlighted, by referencing the `Invoking-Control` of the command.

When the property's value is inherited, for example when a font size property comes from a named style, then the answer must include a discussion of the inheritance, as well as the final place in which the value was set, as in Figure 6. This required a custom answer method in the sample text editor, to generate understandable messages. However, facilities in the Crystal framework automatically traverse the command's `Dependencies` to determine the properties that contributed to the current value. If any of those properties themselves were inherited, then Crystal recursively goes to those properties' commands, and then to their `Dependencies`, etc. At each step, Crystal checks to see if the property is marked as `Show-In-Why-Menus`. If so, another sentence is added to the answer window. (Internal properties are often involved in dependencies, but should not be shown because users cannot change them.) When there are multiple steps, then a "How can I…" question is added to the end of the answer, so the user can ask about each step individually.

To highlight the controls, Crystal needs the ability to bring up widgets programmatically, set them to specific values, find their location, and highlight them, while still having them be operational for the user. Furthermore, the dialog boxes need to keep track of what causes them to be displayed, so Crystal can highlight the appropriate menu item. We were able to implement all of these using the Swing toolkit. Such support is also available in other commercial toolkits such as Mac OS X's Cocoa, where it has been used to implement several types of universal access features.

Implementing the Sample Text Editor

We implemented parts of a sample text editor using the Crystal framework as a test. We used a Model-View design, where the view uses the Java Swing `TextLayout` to format each line. Like Glyphs [4], Crystal's model uses an object for each character that stores the letter and all of its properties (font, size, bold, etc.) except location, which is handled by the layout. Along with the regular characters, the Crystal editor adds special invisible markers to show where various operations occurred, such as deleting text. A marker moves with the characters to its left (if any), and can never be deleted (although the question mechanism could decide not to include old deletions in the "why" menus). Styles are implemented as objects with sets of properties that can be inherited by characters. There are no additional structures needed for words or paragraphs in Crystal. About 10% extra code (most of it quite simple) was needed to add support for answering why questions to the text editor.

Other Kinds of Applications

We believe it would be straightforward to use the Crystal framework to implement other types of applications. We chose to implement a text editor because it seemed like the most difficult. For a drawing editor like Microsoft PowerPoint, each graphical object would have a list of user-settable properties and Crystal would automatically keep track of which commands set them. For "smart" operations, such as the automatic adjustment of font sizes, and moving of attached lines when boxes are moved, the developer would add extra commands to the command list to explain why these happened. When the user hits F1, the system should return all objects under the mouse, including individual objects, groups, and background ("master") objects, and put these into the first-level menu. An implementation for spreadsheets might combine the techniques described here with techniques discussed elsewhere [21] [1] that explain how the values were calculated.

USER STUDY

A small lab study was performed to evaluate whether the "why" menus in Crystal were usable, and to what extent they helped users understand what was happening in their user interfaces.

Experimental Setup and Participants

We used a between-participants design, because the key issue is learning about how to use the system. One group used the Crystal sample text editor as shown here, and the other used the identical text editor, but with the "why" menu removed, and F1 disabled. Each group contained 10 participants, all between the ages of 18 and 53 with an average age of 24. 12 participants were male and 8 female. We recruited participants who reported "little or no" experience with Microsoft Word, although they all had extensive general computer experience, and all but two had experience with other text editors. Those two happened to both be in the group with the "why" menus. Participants were randomly assigned to one of the two groups and were paid to participate. The experiment was conducted on a laptop and was recorded.

Both groups received the identical six tasks. These were derived from real observations of Microsoft Word users, published articles about difficulties with Word, and an inspection of Microsoft's support pages. The tasks represent common issues that real Word users encounter. In sum-

mary, the tasks were (1) turn off automatic capitalization; (2) turn off automatic spelling correction; (3) change paragraph formatting; (4) explain why the "Paste" menu item is grayed out; (5) use the Styles mechanism to change italics of some headings; and (6) use the inheritance property of the Styles mechanism to adjust the font size of all headings. However, the tasks were not presented this way. We demonstrated a problem or a surprising behavior (or let the user do it), and then asked them to fix it. For example, the experimenter read the following script as the stimulus for the first task:

1. Type in the following sentence "The abbreviation fl. oz. stands for fluid ounce."
2. You notice that the word processor has capitalized some characters for you, but you don't want this to happen.
3. Your task is to make the automatic capitalization not happen again.
4. When you think you're done, type "fl. oz. stands" again to make sure it works.

In order to make the experiment somewhat realistic, we copied Microsoft Word 2003's "Tools" menu and the "Options" and "Auto Correct Options" dialogs that are invoked using the Tools menu (see Figure 1). All of the submenus and the various tabs on each of these were live, so the users would have to search through more places. Both tasks 1 and 2 required using the "Auto Correct Options" dialog (Figure 1), and no task required using the Options dialog. Tasks 3, 5 and 6 required using the paragraph styles dialog (Figure 6).

The dependent measures were whether the participants were able to complete the tasks at all and how long they took for the ones they completed. A few users got stuck and required hints, and then we counted them as unsuccessful. We were also interested in usability observations.

Results

Because not all participants completed all tasks successfully, the data could not be analyzed using a standard repeated-measures ANOVA. Therefore, we analyzed both the number of tasks completed and the mean time per completed task using between-participants ANOVA. Participants in the "why" menus condition completed an average of 5.60 (93%) of the tasks whereas those without "why" menus completed an average of 4.20 (70%) of the tasks ($F[1, 20] = 12.60$, $p < .005$). As shown in Figure 10, participants with "why" menus had an advantage in each of the six tasks.

Figure 11 shows the average time per task for those participants who could finish it. Participants with "why" menus completed each task in an average of 91.38 (SD = 51.66) seconds, whereas those without "why" menus required an average of 137.74 seconds (SD = 49.62). This difference approached significance ($F[1, 20] = 4.19$, $p = .06$).

The anomalous value for task 6 seems to be due to a few participants in the "without" group accidentally figuring out a workable strategy during task 5, compared to the "why" menu group who almost all used the "why" menus to try to learn how inheritance works.

The participants who saw them really liked the "why" features. Each of the statements got an average agreement value of greater than 6.2 out of 7: "I understand how to use the Why feature in Crystal", "I found the Why feature easy to use", "The Why feature improved my word-processing experience", "The answers provided by the Why feature were easy to understand", "The answers provided by the Why feature were what I wanted to know", "I was comfortable using the Why feature", and "I would really like a Why feature like this in the programs I use."

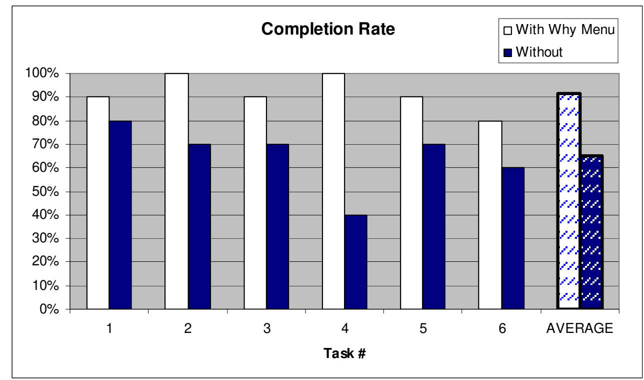

Figure 10: Percent of people in each group that completed the tasks and the overall average. Taller bars are better.

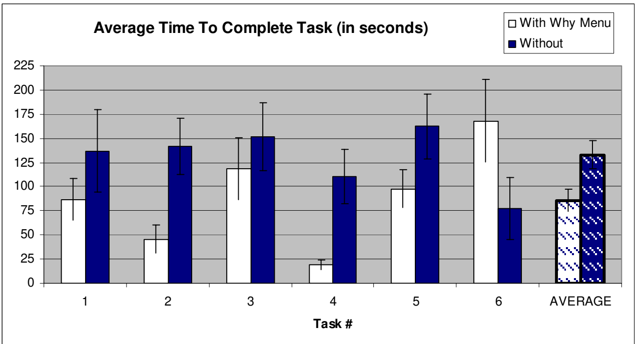

Figure 11: For the participants who could complete the task, the average time they took, with bars showing the standard error of the mean. Shorter bars are better.

Discussion

Clearly, the "why" menus were helpful to users. It is not surprising that the later tasks fared worse, since these tasks were quite difficult, even for some experts. For some people, the "why" features played the crucial role of explaining the concept to some of the participants, which directly led to successful task completion. However, Crystal is not necessarily designed to serve as a tutorial, and it probably did not teach participants about the concept of inheritance if they did not know it already.

We had a number of usability observations about the system. Most of the participants preferred using the F1 key to have more control over the questions they could ask. It

seemed that the most efficient people used the F1 key first. Some participants were reticent to use the F1 key—this apparently was not a natural interaction for them. They used the "Ask about a location…" item in the "why" menu when the desired question was not in the "why" menu directly.

Participants using the "why" features generally knew which objects they should ask questions about, and the questions that showed up matched their expectation. A lot of trial-and-error clicking of menus happened for participants who did not have the "why" features, while the "why" people did not, and seemed to be more purposeful and effective.

FUTURE WORK AND CONCLUSIONS

An obvious next step for Crystal is to do a more complete implementation of the framework so full applications can be built with it, to verify that the ideas scale up and work well in different domains. It would be useful to be able to field-test applications supporting the "why" menus to see to what extent they really help in practice. It would be interesting to see if the Crystal framework would be easier to implement on top of a toolkit with a constraint system such as Citrus [12]. Another open question is how important it is to save the Why information across sessions, so that later users can ask questions about the contents of files read from the disk. We know of no system that saves the undo or command history with the files. The current framework cannot answer questions about operations that are no longer part of the command history.

Everyone to whom we have described the ideas in Crystal has remembered situations in which they wished they could have asked their applications and operating systems why things happened. As even more sophisticated and "intelligent" operations are increasingly added to future systems, asking why will be even more important. Even if natural language processing were to become successful, making the need for Crystal's popup "why" menus unnecessary, the Crystal architecture would still be useful for collecting and organizing the needed information.

ACKNOWLEDGMENTS

Thanks to Susan Fussell for extensive help with the statistics for this paper. Thanks also to Jake Wobbrock and Andrew Faulring for help with this paper. This work was partially supported under NSF grant IIS-0329090 and by the EUSES Consortium via NSF grant ITR-0325273. Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the NSF.

REFERENCES

1. Abraham, R. and Erwig, M. "Goal-Directed Debugging of Spreadsheets," in *VL/HCC 2005.* 2005. Dallas, TX: pp. 37-44.
2. Berlage, T., "A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects." *ACM Transactions on Computer Human Interaction*, 1994. 1(3): pp. 269-294.
3. Bothell, D., *ACT-R Environment Manual.* Version 5.0, April 22, 2004. http://act-r.psy.cmu.edu/software/EnvironmentManual.pdf
4. Calder, P.R. and Linton, M.A. "Glyphs: Flyweight Objects for User Interfaces," in *UIST* 1990. Snowbird, Utah: pp. 92-101.
5. Dourish, P. "Accounting for System Behaviour: Representation, Reflection and Resourceful Action," in *Proceedings of the Third Decennial Conference on Computers in Context CIC'95.* 1995. Aarhus, Denmark:
6. Dworman, G. and Rosenbaum, S. "Helping users to use help: improving interaction with help systems," in *CHI 2004 Extended abstracts.* Vienna, Austria: pp. 1717-1718.
7. Halsted, K.L. and Roberts, J.H. "Eclipse help system: an open source user assistance offering," in *SIGDOC 2002: Proceedings of the 20th annual international conference on Computer documentation.* Toronto, Ontario, Canada: pp. 49-59.
8. Kelleher, C. and Pausch, R. "Stencils-based tutorials: design and evaluation," in *CHI 2005*. Portland, OR: pp. 541-550.
9. Kiczales, G. "Towards a New Model of Abstraction in Software Engineering," in *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures.* 1992.
10. Knabe, K*., et al.* "Apple guide: a case study in user-aided design of online help," in *CHI 1995 Conference companion.* Denver, CO: pp. 286-287.
11. Ko, A.J. and Myers, B.A. "Designing the Whyline, A Debugging Interface for Asking Why and Why Not questions about Runtime Failures," in *CHI.* 2004. pp. 151-158.
12. Ko, A.J. and Myers, B.A. "Citrus: A Toolkit for Simplifying the Creation of Structured Editors for Code and Data," in *UIST* 2005. Seattle, WA: pp. 3-12.
13. Lin, J*., et al.* "The role of context in question answering systems," in *CHI 2003 extended abstracts*. Ft. Lauderdale, FL: pp. 1006-1007.
14. Myers, B.A., *The Case for an Open Data Model.* Carnegie Mellon University, School of Computer Science Technical Report, CMU-CS-98-153, August, 1998.
15. Myers, B.A. "Scripting Graphical Applications by Demonstration," in *CHI 1998*. Los Angeles, CA: pp. 534-541.
16. Myers, B.A. and Kosbie, D. "Reusable Hierarchical Command Objects," in *CHI 1996*. Vancouver, BC, Canada: pp. 260-267.
17. National Instruments, "LabVIEW. National Instruments Corporation, 11500 N Mopac Expwy, Austin, TX 78759-3504," 2005.
18. Nielsen, J. and Molich, R. "Heuristic evaluation of user interfaces," in *CHI 1990*. Seattle, WA: pp. 249-256.
19. Norman, D.A., *The Design of Everyday Things.* 1988, New York: Doubleday.
20. Ramachandran, A. and Young, R.M. "Providing intelligent help across applications in dynamic user and environment contexts," in *IUI* 2005. San Diego, CA: pp. 269-271.
21. Ruthruff, J.R*., et al.*, "Interactive, Visual Fault Localization Support for End-User Programmers." *Journal of Visual Languages and Computing*, 2005. 16(1-2): pp. 3-40.
22. Sukaviriya, P. and Foley, J.D. "Coupling A UI Framework with Automatic Generation of Context-Sensitive Animated Help," in *UIST* 1990. Snowbird, Utah: pp. 152-166.
23. White, R.W., Ruthven, I., and Jose, J.M. "Finding relevant documents using top ranking sentences: an evaluation of two alternative schemes," in *SIGIR '02: The 25th annual international ACM SIGIR conference on Research and development in information retrieval.* 2002. Tampere, Finland: pp. 57-64.