Answering Why and Why Not Questions in User Interfaces

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” system provides an architecture and interaction techniques that allow the user to ask a wide variety of questions about why the actions 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” so “the system should always 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 the 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 all the windows went.

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, then there is no mechanism to figure out why the actions happened, or how to control the actions. 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 a system (see Figure 1) that answers questions about an application. The system we created is called Crystal, which provides Clarifications Regarding Your Software using a Toolkit, Architecture and Language. The idea is that the system makes things “crystal clear.”

Submitted for Publication.
this point Crystal is primarily a feasibility demonstration, but it does show that a system can helpfully answer “why” and “why not” questions for users.

Norman discusses two important “gulfs” in people’s use of their systems [19]. Many help systems are designed for the gulf of execution: teaching users how to perform actions. 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 determining how to fix it if it is not what they intended.

Crystal does not support natural language. Instead, it builds question menus dynamically based on the current state of the application. The user can ask questions either by hitting the F1 key while the mouse cursor is over an item of interest, as was done in Figure 1, in which case Crystal builds a menu of questions about the objects under the mouse. Alternatively, the “Why?” menu can be used to ask about any of the recent actions that did or did not happen.

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 text editor which has some automatic transformations like Microsoft Word.

Crystal automatically provides built-in support for a variety of interaction techniques with which the user can ask questions about their applications. Hitting a key (currently F1) while the mouse cursor is over anything in the interface will ask about objects under the cursor. This includes why menu items are grayed out, why a character has its current formatting, or why an automatic transformation (like spelling correction) did or did not happen on text. 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 as the white space around paragraphs.

After hitting the key, Crystal builds a menu of possible questions about the location. In the text editor, this might include questions about the formatting of a character (“Why is this bold?”), about the word (“Why wasn’t this autocorrected?”), and about the paragraph (“Why is this paragraph double-spaced?”). This list is built automatically from the list of command objects that affected the location. The user can then choose the desired question from the menu.

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 user can choose an action they want to know about. The 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 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 the Crystal also differs in that it automatically determines how to enable the actions.

The Crystal architecture 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!

A small user study suggests that Crystal is effective in teaching users about these complex features, and the interaction techniques were easy to use and well-liked. Subjects with the “why” features were able to complete 30% more tasks than those without, and of the tasks completed, subjects with the “why” features were 21% faster.

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

**RELATED WORK**

Help systems for interactive applications have been extensively studied (e.g., [7] [13] [20] [22]) but most of the systems are designed to help users learn about a command they already know the name of, or learn how to perform tasks, instead of trying to explain situations after they happen like Crystal. For example, Cartoonist displays animated help showing the steps required, but it must be explicitly given the name of a command or task.
Many recent help systems focus on giving tutorials for how to use a system. For example SmartAidé [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 effectiveness of queries that use natural language, which Crystal avoids by presenting popup menus containing specific relevant questions generated from the user’s actions.

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 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 dynamic program execution history.

In its answers, Crystal highlights the actual widgets of the interface. This approach has been used previously in Apple Guide [10], Stencils [8], and the “Show me” feature of some modern help systems.

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’ edits.

**USER INTERFACE**

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 the first issue, we designed the interface to the “why” system with just two interaction techniques to learn. The first is hitting the F1 key while the mouse is over any object. This technique has been used in a few other systems (e.g., Eclipse [7]). The second is using the “Why?” menu to find out about any of the last few actions executed. The “Why?” menu will also contain an item to go into a mode to invoke position-based questions, in case the user does not know 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.

**Questions**

The next important design issue is what questions belong in the menus? If there are too many questions in the menu, then it will take too long for the user to find the desired question. If there are too few, then the desired question may not be there. In general, any explicit user actions should be added, along with any situation where there is a hidden state or invisible dependency. 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, is displayed or not.

For the “Why?” menu, we decided not to add regular typing to the questions, 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, 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).

We do however, provide 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 might have thought would have happened. Therefore, we add to the menu questions for keystrokes that have no effect, such as typing Control-C with nothing selected (see Figure 2). We also add to the menu questions about operations that would have happened if they had not been 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).
Many other things that did not happen are also not added to the “Why?” menu. We do not, for example, add items about 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.

The menu generated when the F1 key is hit will list questions about the objects that the mouse cursor is over. 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 character properties, and Crystal displays all the properties of that character.

The questions in the menus are designed to feature the values in an easy-to-find place (at the end of each line) so that a quick scan will show all the properties’ values. To display each value, we use a variety of rules so the menus are concise yet readable. For Boolean properties, we use the value name or “not” the value name, such as “bold” or “not italic”. For numeric properties, we use property = value. For styles, we just use the style name (such as “Default” in Figure 3).

If the F1 key is hit while the mouse is over a blank part of the window, Crystal adds to the menu questions about why that white space is there, typically based on paragraph properties. Figure 4 shows that an additional question about whitespace is also added, which summarizes all the different contributions to that whitespace (since character and paragraph properties might both be involved).

For any operations that did not happen that are put into the “Why?” menu, the same questions will appear in the F1 menu for the affected objects. For example, the menu that pops up if the user hits F1 over the “Teh” in Figure 2 includes a question about why it was not auto-corrected.

Answers

Answers to the questions typically have two parts: a textual explanation, and highlighting of relevant user interface controls. 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 refer to specific actions that users took or can take by both describing them and highlighting them on the screen (see Figure 5).

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 that dialog. All dialogs are “live” while they are highlighted, so the user can operate them normally. Typically, this will save the user a number of steps if the property needs to be changed. In fact, sometimes we have found it to be quicker to use the F1 feature to get to the desired dialog box compared to finding it among the menus, even when we know 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, we also add a textual explanation, which may be necessary in some situations, and it is useful when the user wants to learn how the application works in detail. We have tried to make the text in the answer window as concise and readable as possible, but it can still get somewhat long, especially when the reason for something is complex. For example, Figure 6 shows the answer explaining why the text is size 20, which is inherited from its style.

Note that because there are multiple causes and actions that the user might want to take, the bottom of this answer window contains links to multiple actions, which in turn, highlight the appropriate controls in menus and dialog boxes. If the user traverses one of these links, the Back button in the answer window is useful for returning to the original question. When the user closes the answer window, the highlighting is removed from the controls.

ARCHITECTURE

An important goal for this research is to create an object-oriented framework that makes it easy to create applications that support asking of “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. Crystal adds specific framework support for application objects and their properties, and commands 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 architecture to implement a small text editor as a test application. We chose a text editor because it is a particularly difficult kind of application to build (as opposed to graphics editors, as have been used to test most previous frameworks [2, 16], which are easier to implement). Also, the Microsoft Word text editor contains many complex features that we wanted to see if our system could help explain.

A clear issue was to determine what information would need to be stored to enable the asking of questions. We decided to store all the information about what happens, and use heuristics to reduce the number of questions and answers shown to the user, instead of throwing away data that may be useful later. This is the same strategy as the WhyLine system [11], which stores every variable setting and method of a program’s execution. Since Crystal only needs to explain user-visible actions, such detailed internal information is not necessary. We found that Crystal does not need to store much more than would be necessary to support undo.

Hierarchical Command Objects

Crystal uses a “Command Object model” [2, 16] to implement all of the actions. There are two primary kinds of commands in Crystal. Action-Commands implement the Java Swing “Action” interface and are attached to the various controls. For example, a Change-Size-Action-Command would be associated with the font size control at the top of Figure 5. Change-Commands perform the actual work in the application, for example, a Change-Character-Size-Command.

Each command object contains a Do-Method that performs its action. The Do-Method of an Action-Command creates the corresponding Change-Command. The Do-Method of a change command performs the change, for example, changing the font size. The change command saves the particular Object-Modified by the action (for example, the characters...
operated upon). Change commands have other methods to **undo** and **redo** the change. Action command objects will contain an **enabled** property to determine if it can be invoked now (whether associated menu items should be grayed out or not). Since Crystal does not use a constraint system like Amulet [16] or Citrus [12], the enabled property is not computed automatically, so it must be set when needed based on information such as when the selection changes.

All command objects will have a constant **label** or a method to generate a label to be used in the undo menu for describing the command. For example, undo “Font Size Select”. The same label is also used for the command by Crystal in the “Why?” menu.

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.

The Crystal architecture adds a few new fields to the command objects specifically to support the asking of the “Why” questions, as well as additional information to the properties of all the user interface objects. Crystal’s command objects also contain:

- **Properties-Used**: 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 Properties-Used 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 key, a toolbar button or a menu item). The Invoking-Control value is used to highlight the control in red as part of answers.

- **Questions-Method**: This method generates the questions that corresponds to the command and will be shown in the menus.

- **Answer-Method**: This method generates the answer using all of the above information. The algorithm is described in more detail later.

- **Undoable/Undone**: In order to keep of history of everything that happens, Crystal saves all executed commands, so those which are saved which cannot be undone are marked. Crystal also records whether the command 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 an action command’s **enabled** property specifies that it is not executable, but it would otherwise have been executed, then a Change-Command is created anyway and put onto the command list, and the Enabled property of the Change-Command is set to false to show that the command was not actually executed. This can happen because the user typed a shortcut key such as Control-C for Copy when nothing was selected, or because an option was disabled, such as turning off auto-correction. 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 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 becoming more common in many 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 need to keep a complete history of all previous actions, even if they were undone, so nothing is ever popped off the command history. 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 already undone, so future undo commands will know to skip over it. Similarly, redo adds a Redo-Command to the command list with an appropriate sub-command, and clears the undone flag from the associated command.

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 and undo history, 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. 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 value will be the 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. We mark internal properties like Font-Size-Inherited with Show-In-Why-Menus set to false so they will not be made visible to users in the “why” menus. We also add to the properties a method which returns a human-readable string, used for displaying the value or the property name and value in the “why” menus. This method implements the heuristics discussed above in the user interface section for displaying values. 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, Crystal adds a special non-undoable command to the beginning of the command list which represents all the default, initial values. Then no additional mechanism is needed for these properties. Figure 1 shows the display for such a default value. For systems such as Microsoft Word where initial values can come from many different places: such as various options, Normal.dot, etc., we envision there being multiple initialization command objects, so each can describe how the user would change the corresponding default value, if desired.

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 history. Note that it will often include more than what would be available in the undo history, since un-executed 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 Question-Method of each command generates the string using 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-Acceptable interface, and if so, calls it. There are three basic ways this interface is implemented.

The first is used when F1 is hit on a Swing control, such as a menu item, and it returns the Action-Command associated with the control. The Questions-Method of the Action-Command will check the Enabled property to see if it should generate a question about why the control is disabled. The Questions-Method might return more than one question: one for why the command is disabled, and another more general question about what the command does.

The second way is used for objects that have properties. In this case, the object is added as a top-level question item, and all the user-visible properties of the object, along with their current values, are added in a sub-menu, as shown in Figure 3.

The third way is used for describing why graphical objects were created or deleted. All graphical objects have a pointer to the command that created them. If this command is marked to Show-In-Why-Menus, then it will be added. For example, the “Paste” command is displayed in the menu, but the typing of regular text is not. Auto-correction is actually implemented as a special kind of create, so a question about auto-correction will be displayed for the appropriate text. 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 Crystal text editor, the objects are invisible markers that flow with the text (see Figure 7).
If the mouse is over whitespace, then Crystal finds all the objects that have contributed to generating that whitespace and adds them to the list, and also adds a special extra question that mentions 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

When the user selects a question, Crystal uses the Answer-Method of the command to generate an answer. For some commands, the answer will just be a constant string. For properties of objects (e.g., “Why is the ‘g’ not bold”), the answer for why it has its current value can be answered 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 in the menus originally 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. The Answer-Method uses the information in the command’s Properties-Used 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 Properties-Used, etc. At each step, Crystal checks to see if the property is marked as user-visible. 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 hack all of these into the Java Swing toolkit. Such support is also available in other commercial user interface toolkits such as Mac OS X’s Cocoa toolkit, where it has been used to implement several types of universal access features.

The Crystal Text Editor

The Crystal Text Editor is implemented using the Crystal framework using 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 keep track of which commands set them. “Smart” operations, such as the automatic adjustment of font sizes, and moving of attached lines when boxes are moved, would insert extra commands into 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 Subjects

We used a between-subjects design, because the key issue is learning about how to use the system. We had two groups, one used the Crystal 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
subjects, all between the ages of 18 and 53 with an average age of 24. 12 subjects were male and 8 female. We specifically only allowed subjects who reported that they had “little or no” experience with Microsoft Word, although they all had extensive computer experience in general, and all but two had experience with other text editors. Those two happened to both be in the group with the “why” menus. Subjects 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: (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; (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.

In order to make the experiment somewhat realistic, we exactly 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 subjects 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 about their use of the system.

Results

Figure 8 shows the percent of the subjects who completed each task. Overall, 92% of the tasks (55 of 60) were completed by the subjects with the “why” menus, compared to 65% of those without (39 of 60). A nonparametric z-test shows these proportions to be statistically significant ($z=3.545$, $p<.05$). Figure 9 shows the average time for those subjects who could complete the task. The overall average for all tasks was 86 seconds with the “why” menus, compared to 133 seconds without. The anomalous value for task 6 seems to be due to a few subjects 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 subjects 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.”

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 subjects, which directly led to successful task completion. However, Crystal is not necessarily designed to serve as a tutorial, and it probably did not teach subjects 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 subjects 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 subjects 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.

Subjects 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 subjects 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

The 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

REFERENCES