Master's thesis

Context-sensitive help as an integral part of a user interface design environment

J.J. de Graaff

Delft, October 1992

Graduation committee:

Prof. Dr. H.G. Sol

Ir. C.A.P.G. v.d. Mast

Prof. Dr. James D. Foley

Ir. B.R. Sodoyer

Delft University of Technology
Faculty of Technical Mathematics and Informatics
Department of Information Systems

Preface

I have written this thesis to graduate from the Delft University of Technology. I have done the research for this thesis at the Graphics, Visualization and Usability center at the Georgia Institute of Technology, where Prof. James Foley has been kind enough to accept me as an exchange student. My whole stay in the United States has been a wondeful experience, and I have learned a lot in this period, not just about computing and human-computer interaction, but also about life in general.

I have to thank a lot of people, with the ever present risk of forgetting someone. I'll do my best to make the list as complete as I can.

First of all I want to thank my advisor in Delft, Charles van der Mast. He gave me the opportunity to go to the GVU center, and provided the initial contact with the people there.

Next I would like to thank Jim Foley, for inviting me to do research in the GVU center, and for working with me. I really enjoyed all the brainstorm session we had, where we came up with a lot of cool things.

Also much thanks to Piyawadee Sukavirya for being my advisor during my stay at the GVU center. Thanks for guiding me through the project, and getting me down to reality every once in a while. It's been fun working with you, Noi.

Another round of thanks to the people I worked with during my research: Mark Gray, Martin Frank, Todd Griffith, Ray Johnson, Krishna Bharat, Srdjan Kovacevic.

I really enjoyed all the meetings we had, discussing UIDE, and all our related projects.

Also thanks to the guys who visited the GVU center before me, Daniel Gieskens and Dennis de Baar. They provided me with some much needed information about staying in the United States and working in the GVU environement, and their enthoused reactions have been part of my decision to go there.

A big hand for the people I spent a lot of time with in Atlanta. There are a lot of people who really tried to give me a good time, but Joe W ehrli, Dean Ptaszynski, T om & Anne Gale, the people at Happy Hour, the people from ORGT, and many more.

Also much thanks to Joan Morton for providing me with all the help she could give, and for all the chatting at strange hours.

In the end, I can only thank my parents, who helped me realize this trip.

Table of Contents

Chapter 1

Introduction

Modern graphical interfaces are full of objects, or widgets, with which the user can interact. Some of these widgets are well-known, e.g. buttons, sliders, scrollbars, menus and the like. Usually, the user can interact with these widgets: buttons can be pressed, menu-items can be selected. This is not always true, though. Sometimes the state of the application doesn't allow a button to be pressed, or a menu-item to be selected. Several strategies are possible to achieve this.

First, the actions could just be ignored by the application. Needless to say that this will lead to much confusion for the user of the application. A second strategy might be to hide the widgets until they become available, but this also creates confusion, although less so than with the previous “solution”.

The final strategy is probably the most prevalent, and counters the disadvantages of the two previous ones. With this strategy, widgets which are not available are still displayed, but with a ghosted, or “grayed out” look. An example of this technique is the interface in Figure 1-1, in which the fire button is not available at the moment.

Although certainly more satisfying than the others, this strategy still leaves the user with some questions. For instance, it might be unclear why a widget is disabled. A related problem is that sometimes the user knows that a particular widget will perform the function which she desires. Alas, the widget is disabled, and there is no way to find out how to enable it. The help system presented in this thesis will help the user with these problems. The help system will be able to explain why a widget is disabled, and how it can be enabled.

1.1 Example

To demonstrate such a help system I first introduce a small application, after which I give some examples of the kind of help such a system could give.

The example application is a small cannon simulation. It features two buttons, one to load the cannon with a cannonball, and one to fire the cannon. There is also a switch which turns the safety on or off. The interface is displayed in Figure 1-1.It is obvious that some buttons cannot be used all the time. For instance, the fire button can only be used if the cannon is loaded, and if the safety is off. Also, if the cannon is already loaded it can not be loaded again before it has been fired.

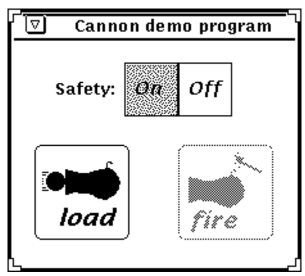

Figure 1-1
Cannon demo interface

Let’s look at an example of help that might be given for this interface. Figure 1-2 shows how the help will be given. In this case the user asked why the fire button was disabled. The help text is shown in the standard OPENLOOK help window , and explains why the fire button is disabled.

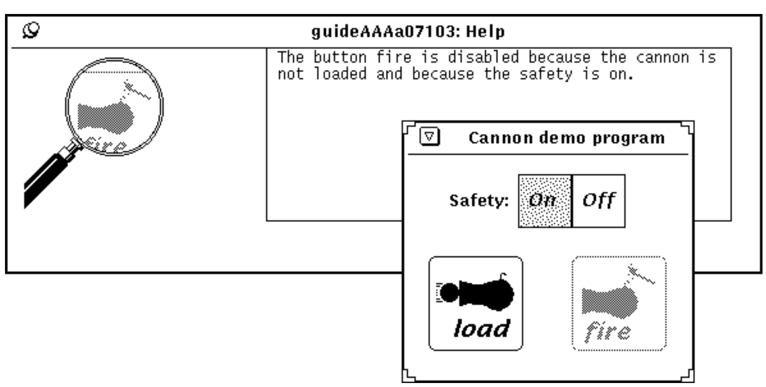

Figure 1-2
Example of a help window

1.2 Focus of research

The focus of this research is to look at the possibilities for the help system described above. It is not my goal to build the help system for this particular application. Obviously, this can be done. In contrast, I am looking for a more general method of generating help systems like this, preferably automatic or semi-automatic. If these help systems can be generated automatically they can be used by applications on a much bigger scale then if they had to be created by hand. Another advantage of generating the help is consistency. Because the help is generated from a description of the interface it will always be consistent with that interface.

This focus implies several questions to be answered by this thesis:

- Can we create the help text automatically?
- If so, how good is the quality, what can be achieved?
- What kind of information do we need to create the help text?

It also implies that I will not look at several other questions, like how friendly the interface to the help system is, how good the wording of the help text actually helps the user, how much the performance of the user is improved by using this help system. That is not to say that these are not important questions. They are, but the focus of this research will be mainly on the technical level. If this research shows that good help text can be generated, then these questions can be researched.

1.3 The remainder of this thesis

Before I introduce the actual help system I want to provide some background information on several areas, which we will need further on. Chapter 2 will be devoted completely to this information. In the first section I will give a brief overview of the research into help to show the current state of research, and the different issues important in this field. After that I will describe a taxonomy of help systems, which allows us to look at the different and independent components of a help system. Using this taxonomy it is possible to discuss small components of the help system separately, instead of discussing the complete system at once.

The next section in Chapter 2 discusses knowledge based user interface software tools. My help system is part of a suite of such tools, `UIDE`. I will describe the structure and concepts of `UIDE` as an example and a reference base.

Finally, Chapter 2 contains a section on planning. To generate the plans the help system is going to present there has to be some planning algorithm, and Section 2.4 will provide both a brief overview of the research in planning, and a description of the planner currently used in the help system.

After this chapter with background information I will describe my help system in Chapter 3. I will do this using the taxonomy presented in Section 2.2, because this allows us to look at each part of the help system in turn, without getting lost in too much information. I will look at each part of the taxonomy in turn, both presenting and defending my choices. In addition to the taxonomy I will also talk about the design process.

Chapter 4 is a chapter about implementation, and it tries to point out several things characteristically for the implementation of the current help system. In Section 4.1 I will take a brief look at the interface system on top of which the help system is build. Section 4.2 describes the structure of the help system, and Section 4.3 mentions some problems with the current environment the help system is working in.

Finally, Chapter 5 contains some conclusions about the help system and my experiences building it. It also contains some discussion about future work in this area: combining text and animation, and making the help system adaptive.

1.4 The research environment

Before we start of with the real work I would like to describe the research environment I have been working in. This might help understand the rest of this thesis a little better, since it will explain the direction and main focus of the center.

I have done this research at the Graphics, Visualization and Usability center at the Georgia Institute of Technology in Atlanta, Georgia in the United States. This center is headed by Prof. Dr. James D. Foley, a well-known user interface researcher. As the name of the center suggest this center focuses on graphics, visualization and usability, both from a technical and a psychological point of view. To be able to achieve these goals the center is multi-disciplinary, and therefore consist of people from several departments at Georgia Tech, like computer science, psychology and industrial and systems engineering.

The Delft University of Technology and the Georgia Institute of Technology have been working together for some time now, cooperating in each other's research programs, and sending students. I have been working there from October 1991 until July 1992. My research at the GVU center will be continued by another student from Delft.

Chapter 2

Problem analysis

In this Chapter I will provide some background information which we will need in the rest of this thesis. Section 2.1 gives an overview of help research and current implementations. Section 2.2 describes a taxonomy of help. With this taxonomy it becomes possible to look at the different components of a help system separately. Section 2.3 describes a knowledge-based user interface system, `UIDE`. This system defines the framework for my research. Section 2.4 looks at planning, some of its history, and the planner which is used in the help system. Finally, Section 2.5 draws some conclusions from all this material with respect to the help system.

2.1 Help: an overview

It is always hard to provide a small overview of a research area, and even more so if the area is as multi-disciplinary as help. In this section I will try to give an overview of help systems and research into this area.

2.1.1 Help as a research topic

A search in the library on help doesn't reveal a lot of books. It seems like help doesn't get much attention in this respect. There are of course some exceptions, one of which even presents a good overview of the research being done in several areas [Kearsley, 1988], but good books on help are few and far between. There are many books that deal with help, but usually as a byline, or only looked at from a certain focus. These books have a different main focus like hypertext or interface design, and they mention help along the way.

So why is it that there isn't much literature on help? My best bet is that it is because help is a very multi-disciplinary topic. It involves a lot of different people. There are papers on help, but again they usually focus on just one aspect of help. Interface designers look at the access methods of help, while cognitive psychologists look at the information presented. Technical writers look at the wording, while artificial intelligence people try to come up with intelligent help systems.

Fortunately, interdisciplinary research laboratories are getting more proliferated, which will provide for a better cross-over between the several disciplines concerned with help.

2.1.2 Quick overview of help research

Part of research into help has been through experimental results [Borenstein, 1985]. These studies have looked at the effectiveness of help on novices and experts, user versus system-initiated messages, static versus dynamic help, and on-line versus off-line messages. Experiments have showed that the contents is much more important than the way help is presented. People will take a bad interface for granted, as long as they are really helped with it.

Research in cognitive psychology has identified a number of basic principles of human behavior that can be used to determine what kind of help may be required. An example of this is the modelling of cognitive processes, for instance with the GOMS model [Card et al., 1983]. Studies of human error [Norman, 1988] also indicate the kind of help needed in a system. The study of mental models created by users about system can also be useful in determining the kinds of problems users might encounter.

Research on individual differences indicates that some dimensions of ability and personality appear to affect how well people use computers. This suggests that individual differences should be accommodated by allowing users to customize interface parameters, including help systems.

Artificial intelligence focuses on the development of help systems that understand what the user is doing and what kind of help is needed. Many researchers believe that natural language will be an important factor in these systems [Wilenskey, 1984].

2.1.3 Current implementations

Most programs these days have some kind of help system. Many just present the manuals for their program on the screen, but a few already advertise with context-sensitive help. Too bad this help usually is not really context-sensitive. In these help systems context-sensitive means that the help system knows in which mode the program is, and at which location the user asked for help. Based on this information the appropriate page of canned text is displayed. I would like to call this kind of help syntactically context-sensitive. This means that the help system knows about the location, or context, of the screen at which help has been asked, and it uses this information to display appropriate, but standard, help.

In this thesis I want to talk about semantically context-sensitive help systems. These systems know about the current state of the application, and the possibilities for the user. Based on this knowledge they will present a help tailored to the current circumstances of the user. For instance, if the user wants to copy a block of text in a word processor, the help system has to know whether there is text, and if so, whether there is a marked piece of text. It should help the user in different ways, depending on these conditions [Sodoyer, 1990; Sukaviriya, 1991].

It is evident that a context-sensitive help system has to have a lot of information about the application. The help system has to have knowledge about everything that could make a difference in explaining something to the user. Two solutions for this problem come to mind. First the help system could be very specialized, and tied into an application very intimately. This approach would allow the help system to access all relevant information, and allow for good context-sensitive help.

A second solution counters the main drawback of the previous one: generality. While a specialized help system can be wonderful, it has to be made over and over again, and specialized each time. Clearly not something software developers would like to have. To do this some kind of uniform representation with information about an application has to exist. This ties in with the efforts of people doing research into user interface design environments. These people are also looking for an uniform representation of applications, on which they can base their interfaces. It makes sense to combine the two efforts, and design a context-sensitive help system as a part of an user interface design environment.

2.2 A help taxonomy

To get a better insight into a help system and its components from both the user and the system view Sukaviriya [Sukaviriya, 1991] presents a taxonomy of help. This taxonomy gives a good overview of all the different aspects of help, and the alternatives for each aspect. The taxonomy separates the help interface, the user view, from the underlying internal structures, the system view. Ideally, both should be completely independent, but this is often not practical. A high level overview of the taxonomy is shown in Table 2-1.

TABLE 2-1
An overview of help taxonomy

Help Interface (User View)	Access
	Content
Help System Strategy (System View)	Information Structure
	Architecture

The user view characterizes what the user sees. It is further decomposed into help access methods and help content. The system strategy view characterizes how the system handles the information that is the basis for help, and it looks at the structure of the information and the architecture of the help system. Section 2.2.1 will elaborate the user view, and Section 2.2.2 the system view.

2.2.1 Help interface: the user view

The help interface defines how the user will see the help system. It consists of two main parts, help access and help content. The help access defines who invokes the help, and how the help is invoked. Help can be invoked by the user or by the system. If the user invokes help, this can be done in several ways. The help content defines what the user will experience once the help is invoked. Table 2-2 gives an overview of the user view.

TABLE 2-2
Help taxonomy: the user view

Help Interface (User View)	Help Access	INITIATOR: System-initiated/User-initiated
		ACCESS METHOD: Keyword-based / Natural-language-based / Graphical / Gestural / Spoken
	Help Content	CONTENT: Conceptual / Procedural
		MEDIUM: Textual / Graphical / Animated / Speech Audio / Multimedia
		ADAPTIVENESS: Static / Context-sensitive / History-sensitive / User-sensitive

Initiator

Either the system or the user will initiate the help. System-initiated help is also called active help, while user-initiated help is called passive. Traditionally, help has been passive. Active help is harder to implement, because apart from generating appropriate help the system also has to monitor the user, and assess whether the user needs help. This means the system has to have a very good understanding of the task-domain. So far, these systems are still in the experimental stage. Passive systems are easier to implement because the user indicates that she wants help.

Access method

A passive help system requires the user to indicate the need for help. In traditional textual interfaces this is usually done with keywords or natural language. With the appearance of graphical interfaces the access methods also became more graphical, ranging from selecting a menu-item to manipulating objects. Also, the use of gestural access methods is being researched. For some specific situations sound or spoken language can be used as an access method.

Content

The content describes what kind of information the help will convey. Two different kinds of help can be distinguished, *conceptual* and *procedural*. Conceptual help explains the concepts of the application, i.e. all the knowledge needed to work with an application. For instance, conceptual help explains how different icons look, and what they represent in a direct manipulation interface. Procedural help explains procedures in the interface. It concentrates on the tasks to be performed, and is directly applicable.

Medium

The medium defines how the help will be presented to the user. Traditionally, this has been done with *text*. With the appearance of graphical user interfaces help can also be presented with *static graphics*. Other media, such as *speech* and *animation* are currently being researched, and can be very useful for particular tasks. *Multimedia* help combines several of the media mentioned to communicate to the user.

Adaptiveness

Traditional help systems are considered *static*. They do not adapt under any circumstances, and just reproduce the canned help if needed. If the help is generated on the fly it will be easier for the system to adjust the content to the current context. This results in *context-sensitive* help. Context-sensitive help only looks at the context of the system. It might also want to adapt to the specific user working with the system. Such a help system is called *adaptive*. It will adapt the help it presents based on the user's knowledge, preferences, etc., and the context she is in.

2.2.2 System strategy: the system view

The system strategy defines how the system handles help. It is concerned with the structure of the information used to generate the help, and with the architecture between the help system and its environment. Table 2-3 gives an overview of this part of the help system. To provide adequate help a help system needs to have

TABLE 2-3
Help taxonomy: the system view

<table>
<tr><td rowspan="8">Help System Strategy

(System View)</td><td rowspan="5">Information Structure</td><td rowspan="2">Declarative Knowledge</td><td>Linear Text</td></tr>
<tr><td>Entity-based</td></tr>
<tr><td rowspan="3">Procedural Knowledge</td><td>Rule-based</td></tr>
<tr><td>Grammar-based</td></tr>
<tr><td>State-based</td></tr>
<tr><td rowspan="3">Architecture</td><td colspan="2">INTEGRATION:
Isolated Help Module / Integrated with design tool</td></tr>
<tr><td colspan="2">DESIGN-TIME KNOWLEDGE SHARING:
Separated Help Knowledge / Shared Help Knowledge with UI Tool</td></tr>
<tr><td colspan="2">DOMAIN INDEPENDENCE:
Application-dependent / Application-independent</td></tr>
</table>

domain knowledge. This knowledge can be either *declarative* or *procedural*. Declarative knowledge is usually explicitly stored, because it describes the concepts of the application and interface, and these are unlikely to change as the application is executed. Therefore the main task of the help system is to retrieve all the parts and present them in a useful order. Procedural knowledge is usually implicitly stored, because it depends on the current state of the application. The help system has to generate knowledge by means of procedures, and transform this knowledge into actual help. Both forms will be described in more detail below, followed by a discussion of the several architectural approaches.

Declarative knowledge

Declarative knowledge is knowledge which is stored explicitly. For this reason declarative knowledge is often used for conceptual help, because concepts of an application don't change over time. If the help is stored explicitly it only needs to be recalled to compose a message. This does not necessarily mean that the help is static, though. The stored pieces of help could be assembled into an overall presentation on the fly, thereby making context-sensitive, and even adaptive help systems possible.

The easiest way to store declarative knowledge is to use linear text. Examples of this are the UNIX man pages. A drawback of this method is that the chunks of information are usually rather large, so it is hard to manipulate them. This approach is often used in combination with static help. A better way is to store pieces of information as entities. The help system can string information from several entities together to provide context-sensitive or adaptive help.

The granularity of the knowledge is a good indication about the abilities of this kind of help system. Coarse-grained knowledge can only be the basis for large chunks of stored help. The help system will not have many possibilities to manipulate and rearrange help elements. If the knowledge is fine-grained, it will be possible to combine the different elements in ever-changing combinations, thereby providing far more flexible help.

Procedural knowledge

If the knowledge is procedural then the knowledge about the system is part of the system, or better, the knowledge about the system can be inferred from the system itself. The three most common approaches are rule-based, grammar-based and state-based. These systems differ only in the way they are implemented, the way procedural knowledge is stored is the same.

For instance, in rule-based systems all knowledge of the application is stored in rules, e.g. if this happens, do that, and if that happens, do yet another thing, etc. These rules are an integral part of the application, without them the application would only consist of a framework, but it would not be able to perform any action. A help system could also look at these rules, and infer behavior from them, e.g. it interprets these rules, and infers what the application would do. This information is then used to generate the help.

To summarize, procedural help is based on a description of the procedures or actions in an application, while these descriptions are also used by the application itself.

A drawback of using procedural knowledge is that it usually is very domain dependent, and can not be used for a new application domain, because the help system has to know about the specific rules or routines in the application in order to generate the help.

Help architecture

With the architecture three different issues arise: integration, design-time knowledge sharing and domain independence.

- A help system can be integrated within tools that are used to design the system, or it can be an isolated module.
- A help system can share knowledge with a design tool, or it can have its own knowledge base. Sharing knowledge has the advantage that more information can be used, but it might also restrict the help system.
- A help system can be application-dependent, or not. If the help system is dependant, it can probably provide better help within its specific domain. On the other hand, if a help system is general enough it can be used with different applications without any work on the help system.

2.3 Knowledge-based user interface software tools

In the last few years much research has been done into the area of knowledge-driven software tools. Examples of such efforts are the `UIDE` suite [Foley et al., 1991], `ITS` [Wiecha et al., 1990] and `Humanoid` [Szekely et al., 1992]. At the heart of these systems is a knowledge base with which the application is described in a formal way. This knowledge base is then used to generate the user interface more or less automatically.

In this section I will focus on `UIDE`, the User Interface Development Environment, a suite of tools for designing and maintaining user interfaces [Foley et al., 1988; Foley et al., 1989; Foley et al., 1991]. I give a brief overview, after which I will focus on those features that are interesting with respect to my research.

2.3.1 UIDE: An overview

Figure 2-1 shows the organization of `UIDE`. From this figure it is clear that the knowledge base is a central concept in `UIDE`. All tools that are part of `UIDE` use this knowledge base to store and retrieve information about the application and interface. Because there is only one set of data this data is always consistent, and redundancy is avoided. Also, each tool can access the information gathered by each other tool, which means that each tool can access information it otherwise wouldn't have had.

2.3.2 Action descriptions

At the heart of `UIDE` is a hierarchy of action descriptions. This hierarchy describes the actions and parameters of the application, how these actions and parameters are tied to an interface, and how this interface will behave. Figure 2-2 gives an overview

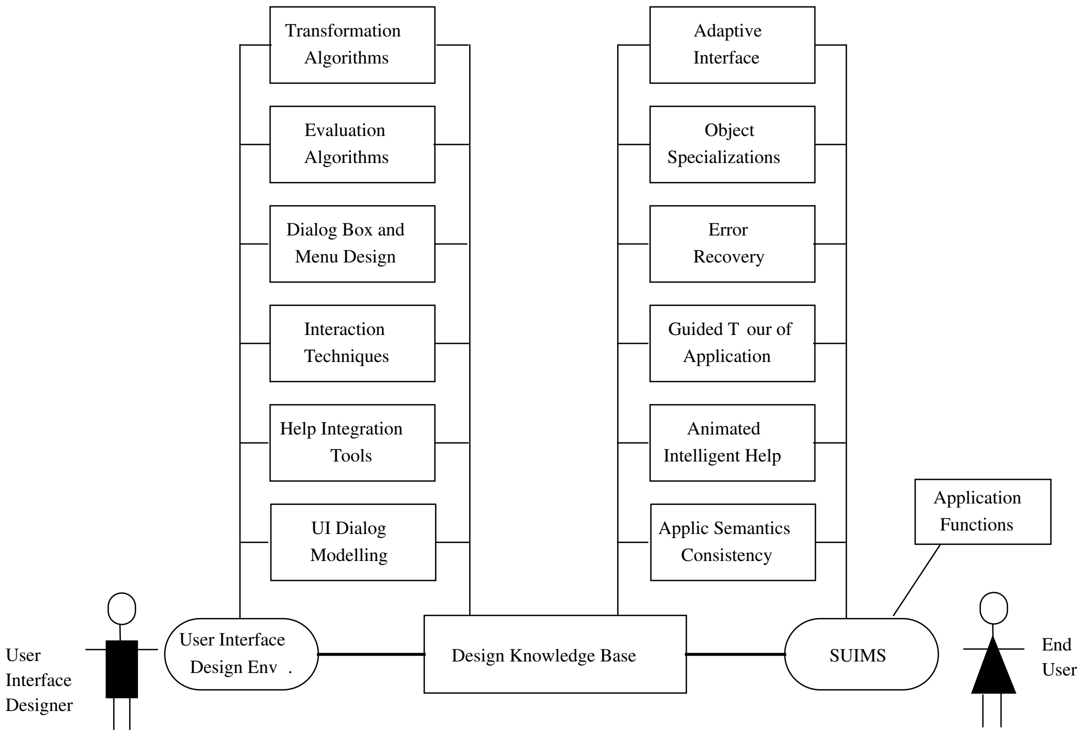

Figure 2-1
UIDE environment

of this hierarchy . There are three different levels of action descriptions. The highest level contains semantic actions meaningful to the application. The next level consists of actions that get values or selections from the interface to the application. The last level in the hierarchy deals with user interactions with the interface.

I will look at each level in turn, and use an example to clarify the differences between the levels. As an example I will describe an application action to rotate an object a certain angle. Figure 2-3 shows the hierarchy for this example.

Application actions

At the highest level application actions and their parameters are described. These actions are interface-independent. They are not committed to any specific interaction technique or interface style. Each application action has pre-conditions, which state what must be true in the context of the application before this action can be invoked. Each action also has post-conditions, which state what changes in the application context when the action is executed.

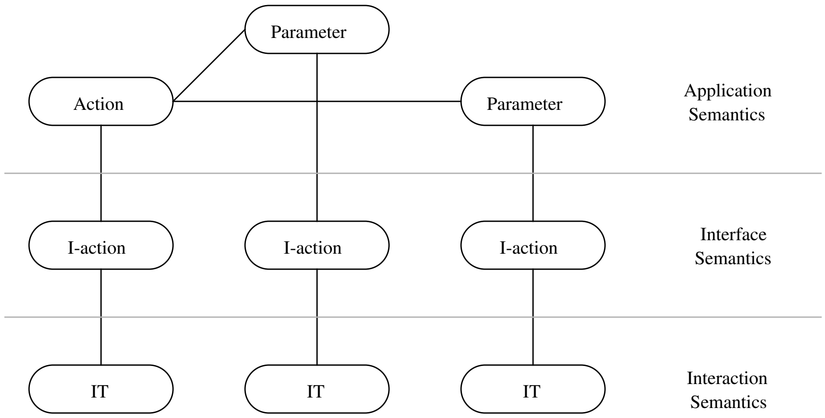

Figure 2-2
UIDE action hierarchy

For instance, the action is rotate-object, and its parameters are object and rotation-angle. A pre-condition is that there must be an object to rotate, and a post-condition would denote the new orientation of the object.

Interface actions

Interface actions, the next level in the hierarchy , tie the application actions to a particular type of interface. At this level the designer defines what kind of interface will be used for an action and its parameters. Each interface action also has pre- and post-conditions. This time, however , they consider the context of the interface as opposed to the application. These pre- and post-conditions also control sequencing, for instance.

In the example I could tie a select-action action to the rotate-action, which means that rotate-action has to be selected to be invoked. The actual method of selection will be deferred to the next level. I will also tie a select-object action to the object parameter , and a enter -integer action to the rotation-angle parameter . The pre- and post-conditions will control the sequencing, so that I can only enter the rotation-angle when an object is selected.

Interaction techniques

The last level in the hierarchy consist of interaction techniques. Interaction techniques specify how interface actions are carried out in an interface. More than one interaction technique can be attached to an interface action, because there can be more than one way to perform that action in the interface. Interaction techniques also have pre- and post-conditions, which tie closely to the screen context.

To complete the example I can now attach the actual selection method to the rotation-action, for instance a pulldown menu item. I tie two different interaction techniques to the select-object action, a mouse-click and a type-in-object. This means the user can both click on an object, or type its name to select it. The complete example is shown in Figure 2-3, the pre- and post-conditions are omitted for clarity.

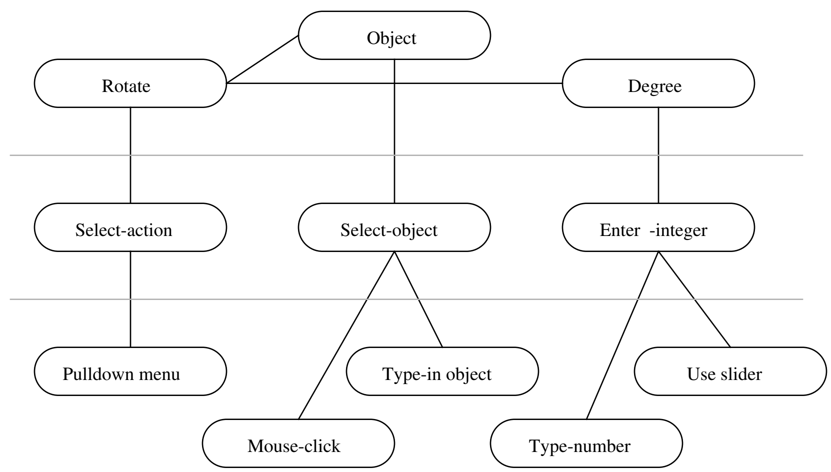

Figure 2-3
Action hierarchy for rotate-object example

2.3.3 Classes in the knowledge base

The information described in the previous section is contained in seven classes of data. The relations between the different classes are visualized in Figure 2-4.

- The *Object* class represents application objects with which the interface has to deal. Information stored in this class includes the name, parent class, subclasses, attributes and a list of action that can be applied to the object.

- The *Attribute* class represents the attributes of the objects. Attributes consist of a name, default value and an attribute type.

- The *Attribute type* represents the data type on an attribute and other information unique to that particular data type.

- The *Action* class represents both application and interface actions. Information stored in this class includes the name, parameters, pre- and post-conditions, a list of objects the action can be applied to, and if possible an inverse action.

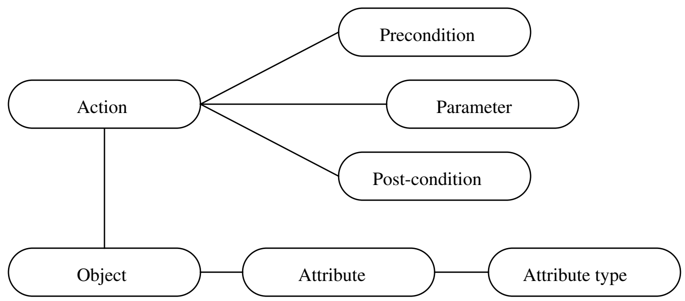

Figure 2-4
UIDE Object classes

- The *Parameter* class is used to store parameters that are passed to the actions.
- The *Precondition* class represents expressions that have to be true to perform a certain action.
- The *Post-condition* class represents assignments that will be true after executing a certain action.

2.4 Planning

If the help system wants to explain how a widget can be enabled, then it needs some kind of planning algorithm to get the information needed for that explanation. Planning algorithms, or planners, are studied as part of the field of Artificial Intelligence, and some good literature is available. For instance, a nice introduction with examples [Nilsson, 1980], some more advanced planners [Sacerdoti, 1977, Wilkins, 1988], and a good overview of the current state of the art [Allen et al., 1990].

Despite the amount of literature, planning is far from a clear-cut problem. Simple and elegant algorithms exist, but they can not cope with all the exceptions and quirks of the real world. This means these algorithms have to be tweaked or changed to include the wanted behavior, making the planners more complex, less general, and less straightforward. The goal of this project is not to do research into planners, and therefore I just adapt a simple planner to get the job done. I give a brief introduction to the field of planning, and identify some characteristics that are interesting with respect to the help system.

2.4.1 Overview

Planning is one of the key elements of AI-research. As early as 1959 people started to design algorithms to aid planning. David Chapman gives a good overview of the developments [Chapman, 1985]. Two planners introduced concepts that are still in use. GPS introduced means-end analysis. With this technique problems are solved by applying an operator to a goal, and listing the pre-conditions of that operator as new goals. STRIPS introduced the action-model, in which each step has post-conditions denoting the changes made by that step in the context. Both these techniques are used by most modern planners.

To be able to use means-end analysis and the action-model both pre- and postconditions have to be present. These can be produced by the action hierarchy described in Section 2.3.2. This means that planning for the action hierarchy will be straightforward, without any conversion.

2.4.2 STRIPS

`STRIPS` is a simple and very elegant planner [Nilsson, 1980]. This simplicity makes it less useful for many applications, but it is still powerful enough to function as a first introduction to planning. `STRIPS` maintains a stack of goals and focuses its problem-solving effort on the top goal of the stack. Initially, the goal stack contains just the main goal. Whenever the top goal matches the current state description, it will be removed from the stack, and the match substitution is applied to the expressions beneath it in the stack. Otherwise, if the top goal is a compound goal, `STRIPS` adds each of the component goals in some order to the stack.

When the top goal is a single-literal goal, `STRIPS` looks for a rule whose add-list contains a literal that can be matched to it. If a match is found the rule is added to the stack, and the precondition of this rule is then added to the stack, becoming a subgoal.

When the top goal is a rule, this means that all its preconditions have been satisfied. This in turn means that the rule is applicable, and it will be applied to the current state description and removed from the stack. Figure 2-5 shows this algorithm in a schematic way.

2.4.3 Improvements

`STRIPS` can be improved upon in several ways. One of the improvements is to make the planner work with hierarchical data. Normally, the planner spends a lot of time looking at all the alternatives at each junction. This can be cut back by using hierarchies. The idea is to mark some pre-conditions as more important than others, so that they can be handled first. If all these high-level pre-conditions are planned

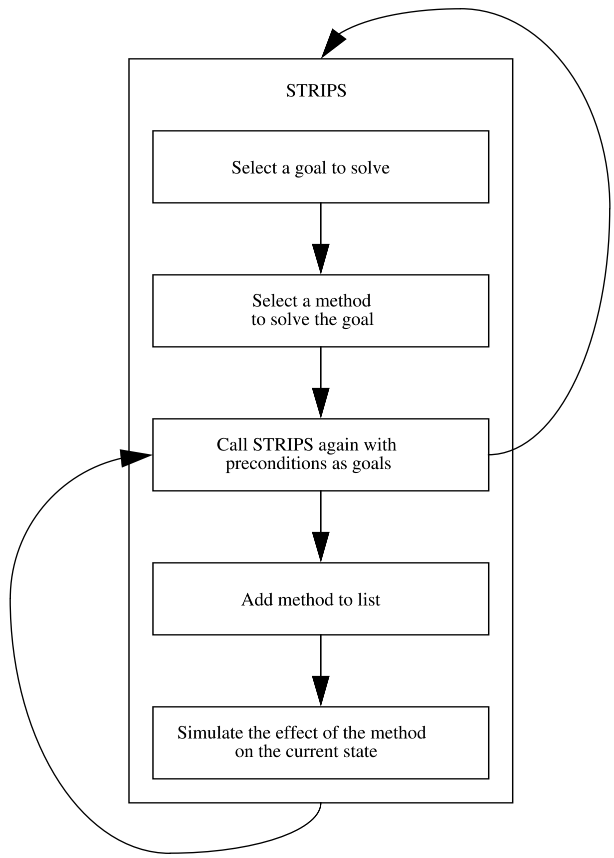

Figure 2-5
The STRIPS planning algorithm

for the next level of pre-conditions will be done. This means that the planner will come up with a global plan fast, and that plan will be refined and more detailed with each lower level of pre-conditions.

This notion of hierarchical planners can make use of the already existing hierarchy in the knowledge base of UIDE. The application action level can be used to make global plans, denoting which actions have to be performed, while the interface action level could detail these plans in terms of the actual interface.

2.5 Conclusions

Let's look at what we have discussed so far, and how it relates to the rest of the thesis. First we have looked at the current state of research and implementation.

This merely provided some background in the field, and this will not be used explicitly in the rest of the thesis. After that we looked at a taxonomy of help systems. This taxonomy allows us to look at the different, and partly orthogonal, parts of a help system. The taxonomy will be used in the next chapter to describe the help system.

After the taxonomy we looked at two other things needed for the help system. The first is the environment the help system will be part of. This environment is `UIDE`, the User Interface Development Environment. Important features here were the existence of pre- and postconditions to control the interface, the fact that this happens on several levels of abstraction, and the notion of a central knowledge base.

We also looked at planning algorithms, which we will need to generate plans to explain how to enable a widget. We saw that most planning algorithms use pre- and postconditions as their input, which can be supplied by `UIDE`. We also looked at a simple planning algorithm, which will be implemented in the help system.

All of this leads us to the next chapter, in which the complete help system design will be described. All the information from this chapter will not be used directly, but it will be present in the design.

Chapter 3

Functional design

In Chapter 1 I described a help system to explain why widgets are disabled, and how they can be enabled. In this chapter I will describe this system in more detail. First, I will give a quick overview of the help system, using the help taxonomy described in Section 2.2 on page 7. This quick overview can be found in Table 3-1, and lists both the chosen approach along with a short justification, and a reference to the section in which the item is discussed in more detail.At the end of the chapter I will also look at the design-process. This part is not included in the taxonomy, but nevertheless an interesting thing. With the design-process I mean the design-process of the interface designer. What does he have to specify, and how does he specify it.

TABLE 3-1
Help system requirements

Help interface User View	Help access	Initiator	The system will be user-initiated.	3.1.1
		Access method	The access method will include pointing to the object about which the user wants information.	3.1.2
	Help content	Content	The context will consist of explanations as to why an object is disabled, and how to enable it. This is procedural help.	3.2.1
		Medium	The help will be presented using text, since this is best suited to present abstractions.	3.2.2
		Adaptiveness	The help presented will be context-sensitive, based on a dialogue model.	3.2.3

TABLE 3-1
Help system requirements

Help system strategy (System view)	Information structure		The information structure will have a low granularity, and it will use both declarative and procedural means to generate the help.	3.3
	Architecture	Integration	The help system will be integrated with the design tool, that is, systems created with the design tool will make use of the help system.	3.4.1
		Design-time knowledge sharing	The help system shares knowledge with the underlying UI tool. The knowledge shared is the dialogue model.	3.4.2
		Domain independence	The help system is application independent.	3.4.3

3.1 Access

The access mechanism has not been part of this research, so this will be a brief section, mainly containing some recommendations.

3.1.1 Initiator

The help system is user-initiated. A system-initiated help system is difficult to build, given the goal of the help system. It is very hard to determine whether a user knows why a widget is disabled, and if she wants to know why right now. Even though this is an intriguing problem, it was not within the research focus, and therefore I implemented a user-initiated system.

3.1.2 Access method

Since the help system exists within a graphical interface, and deals with widgets, the most obvious way to access the help is by pointing to the widget in question, and indicating to the system that help is required. However, different interface environments have different help mechanisms, and I feel that the access method should conform to the environment it is running in.

For instance, the current implementation runs within the `OpenWindows` environment [Sun, 1990]. Within this environment the standard way to get help is to point at a widget, and press the `Help` key. Normally, a window pops up containing some static text about the selected widget. I have changed this to include the help text as well. A better solution would be to include buttons on the standard help window , with which the user would be able to ask for the why and how explanations. This could be extended with shortcuts, so that the experienced user could get the desired help immediately .

3.2 Help content

In Section 1.1 on page 2 I already gave a small example of how the help system could look. In this section I will describe the content of the help system in a more formal and complete way . There are three different issues with respect to help content: contents of the help text, medium and adaptiveness. Of these three, content is the most important one. Research has already shown that contents has a much bigger influence on users than medium.

3.2.1 Contents

In Section 1.1 I already described the content of the help system. It will explain why a widget is disabled, and how it can be enabled.

As an example we will look at a small application. This application is a simulated cannon. Figure 3-1 shows the interface of the cannon, and the pre- and post-conditions associated with each widget.

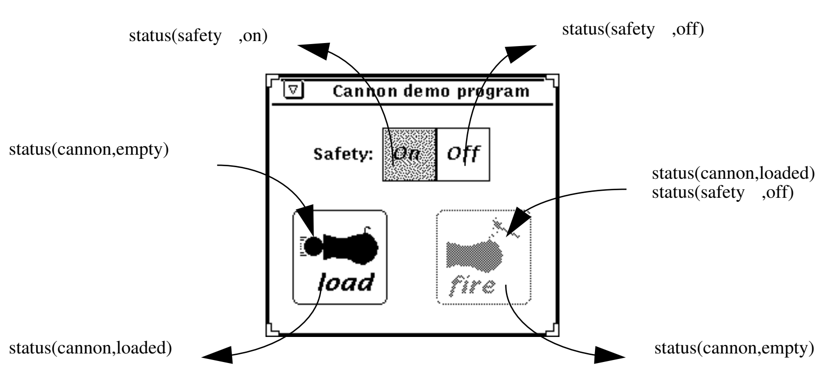

Figure 3-1

Example application with pre- and post-conditions

Explaining why

The easiest way to explain why an object is disabled is to look for the reasons that made the system decide to disable this object, and give these reasons to the user . With this approach the explanation will be based solely on the direct context of the object in question. An example of this is: “The *Fire* button is disabled because the cannon is not loaded.”

One might argue that better explanations might be provided if more information is known. Let’s look at an extreme case where we get all the information possible. An example of this is “The Fire button is disabled because the cannon is not clean.” This approach might confuse the user because this way the help system presents indirect reasons. It is true that the cannon cannot be fired because the cannon is not clean, but this is only an indirect reason. The real reason is that the cannon can not be fired because it isn’t loaded, and it can’t be loaded because it isn’t clean. With larger interfaces this will direct the attention of the user in the wrong direction. It clearly will not help the user.

Maybe it is better to present all the information, as in “The Fire button is disabled because the cannon is not loaded, and it cannot be loaded because the cannon is not clean.” This will surely give enough information to the user, but that is also its weak point. Explanations of this kind tend to get long, too long to be easily comprehended. The user doesn’t want a complete and in-depth review of the relations in the interface, but a quick explanation of the current situation within the local context.

However, I could provide the user with something more than just the reasons for the disablement. After only one step of backchaining the help system can present the widgets that can enable the disabled widget. These widgets can be mentioned in the explanation, so that the user would at least have a clue where to look next, without having to ask a complete plan. An example of this would be: “The Fire button is disabled because the cannon is not loaded. Use the Load button to load the cannon.”

Explaining how

If the help system has to explain the user how to enable an object, then the best way to do this is to present the user with a plan. The user can execute this plan step by step to reach the goal of enabling the object. This seems to be rather trivial, but to problems appear with presenting the plans.

First, long plans are hard to memorize. Even though the user can retain the plans in the current environment, this might not always be the case. Also, lengthy plans may cause the user to loose track of what she is doing, and the problem solving gets reduced to executing a number of mindless small interactions with the interface. This will not learn the user anything about the task at hand, because it prevents chuncking the information.

A solution for these problems is to make use of abstractions. Instead of presenting all the small interactions that make up an action, just the action is presented. This reduces the length of the plan, but it also helps the user to understand how all the actions relate to each other . It is easier for the user to make chunks of actions, because the help is structured that way . As an example think of “First load the file, then start spell-checking” as opposed to “First select the load menu-item, then type in a name, then click the load button, then start spell-checking.”

The second problem with presenting plans is the existence of alternatives. A plan might have several alternatives, which would all result in the goal of enabling the object. For instance, the pause button on a tape player can be enabled both with the play and record button. Some questions have to be answered in this case, for instance, do I present both alternatives? If so, do I treat them equally , or does one get emphasized? Which one?

3.2.2 Medium

Since the help system gives help within a graphical context, graphics as a medium seem very natural. It has the benefit of blending in with the rest of the interface, and thereby providing help in a natural way . Another advantage of graphics is that it can refer to the interface much easier than text. For instance, it is much easier to highlight or point to a widget, than to describe it with text.

Because graphics seem such a natural choice it might come as a surprise that the help system uses text for its explanations. I agree that graphics are powerful, and they certainly have benefits in a graphical interface, but they lack one powerful feature, the ability to express abstract things. This causes some problems when explaining how , although an animation of all the steps needed could “explain” that. I say could, because it is clear that a textual explanation seems to have better long-term effects, while animation is just mimicked and forgotten [Palmiter and Elkerton, 1991].

Graphics will not work if the help system has to explain why a widget is disabled. These reasons can not be represented graphically without much work, both for the designer and the user . For instance, many things can be explained by using cartoons to depict the reasons, but it would take an enormous amount of time to design all these cartoons and the user must interpret them correctly . Text is simply less ambiguous and much more to the point in this case.

3.2.3 Adaptiveness

The help system is context-sensitive. Since it has to explain why certain objects are disabled, and how to enable objects, it has to know about the context of the application. The reasons why an object is disabled can change over time, as the state of the application changes.

The help system will not be adaptive, because I do not take user models into account for the current research. Therefore the help system is unable to know anything about the current user, which in turn makes it impossible to adapt to that user.

3.3 Information structure

In Section 2.2.2 on page 10 I described that the help system uses either procedural or declarative knowledge. In reality these two forms are opposite ends of a spectrum, and hardly any system is completely procedural or declarative. For instance, the help system is mostly procedural, but it also contains declarative knowledge.

The declarative knowledge consists mainly of templates. These templates are stored pieces of text, and they are used to construct the natural language help. The procedural knowledge consists of the pre- and post-conditions that describe the interface and application. They are used to create the actual explanations. The use of templates and pre- and post-conditions will be explained in more detail later in this section.

Granularity is a good measure for flexibility of the help system. If the elements of a help system are coarse-grained, then the help system doesn't have much possibilities in combining the different elements into different help text. If the elements are fine-grained, a nearly infinite number of combinations are possible. The information in the help system is fine-grained, for otherwise it would be impossible to generate ever changing context-sensitive help on the fly.

3.3.1 Declarative information: Templates

The declarative information in the help system consists of templates. These templates are used to present the procedural information in a readable form. To remain flexible in generating the help text I use several levels of templates. First I look at the highest level, which defines the general format of help. Second I look at the templates used to transform procedural information.

Standard format for explaining why

To be able to generate the text automatically I need to have some standard format for explaining why, in which all the different parts will fit. This standard format will have two parts. The first part will explain why the object is disabled, and the second part will indicate other objects that can help enabling the object in question.

The first part will explain why the object in question is disabled. This part will have to contain a reference to the object in question, and a list of all the reasons that cause the object to be disabled. A template might be: "The *object* is disabled because *reasons* ." Further on in this section I will explain how the reasons will be explained. The reference to the object should include both the name and the type of the object, e.g. "the button load", or "the menu-item erase".

The second part will consist of a sentence for each constraining predicate indicating an object that can make the constraining predicate true. If there are more objects that can make a certain predicate true, only one will be presented, e.g. "To *make this reason true* use the *object*. Again, the object will be named with both his type and name.

Here's an example to make the abstract format more clear. "The button fire is disabled because the canon is not loaded and the safety is not off. To get the canon loaded use the load button. To get the safety off use the safety switch."

3.3.2 Procedural information: planning

The procedural part of the available information consists of the pre- and post-conditions that describe the interface. These pre- and post-conditions are used to enable and disable the interface objects based on the state of the application. The help system also uses this information to generate the help text.

To create a textual explanation for a single predicate I will have to look at a way to convert that predicate into text. The most convenient way to do this is to use templates. A template in this context is a standard phrase apt for this particular predicate. The template has some slots in which the arguments of the predicate can be dropped.

An example of this is the *status* predicate. This predicate takes two arguments, the first being the object, and the second the state of that object. A typical status-predicate looks like this: status(Machine, On). The template for the status-predicate would be: "the *argument 1* is not *argument 2* ". Combining the template with the predicate will result in: "The machine is not on", which is a plausible explanation for this predicate.

Although a number of standard predicates can be provided, the help system also allows the designer to specify her own combination of templates and predicates, which will allow for domain-specific explanations.

3.4 Architecture

The architectural issues have mostly been given by the research environment. A set of tools and environments was already given, and this research has made use of the existing framework.

3.4.1 Integration

The help system is integrated with the UIDE suite of tools. It uses a system which controls the user interface with pre- and post-conditions. This system was an earlier project in the UIDE environment [Gieskens and Foley, 1992].

3.4.2 Design-time knowledge sharing

The help system shares knowledge during design-time. This follows from the structure of UIDE, which is described in Section 2.3 on page 12. There is only one knowledge base, which is used by all tools, both during design-time and run-time. The sharing of run-time knowledge follows from the fact that the help system uses procedural knowledge to generate help text.

3.4.3 Domain independence

The help system is domain independent, because it doesn't rely on specific features of an interface. If an application can be described with the hierarchy of actions from Section 2.3.2 on page 12, then the help system can provide help for it.

3.5 Design process

The design process is not described by the taxonomy, but I think it is still important enough to mention. The design process describes how the help system will be used from the designers side, with questions about what to specify and what to edit.

As I mentioned in Section 3.3 the help system uses both procedural and declarative knowledge. The declarative knowledge consists of templates with slots which will be filled in with procedural knowledge.

This means the designer has to work with two different sources of information. Most of the time he will only be concerned with the procedural knowledge, because this describes the actual interface being used. There is a inherent problem when using procedural knowledge. The procedural knowledge is not needed for the help system, but primarily for the actual procedures themselves. In the help system this means that the pre- and postconditions have to be constructed according to certain rules, in order to encode the information in a proper way . This might sometimes be a problem, because when naming the predicates the fact that these names will be used in the help system has to be considered.

The designer will need to spent less time with the declarative knowledge. There is already a set of standard templates that can be used, Only if these are not sufficient will the designer have to create other templates. This might occur when some pre- and postconditions are very application-oriented, in which case they will need some special kind of explanation.

A practical example of generating an interface will be shown in Section A.1, where a complete interface will be build from scratch using the design tool DevGuide.

Chapter 4

Implementation

The previous chapter described the help system in a formal way. We have defined all the possible aspects of the help system. In this chapter I will write something about the actual implementation of the help system. This will help put things more into perspective. First I will describe the interface system on top of which the help system has been build. This interface system is also part of UIDE. Next I will describe the global structure of the help system. Next I will describe some of the particularities about the current environment for the implementation. Finally I will discuss some of the demonstration I will give in the appendices.

4.1 The interface system

The help system has been build on top of another UIDE project by Daniel Gieskens. This system allows the user interface to be controlled by pre- and postconditions. To be able to do this each widget in the user interface has pre- and postconditions. A special widget manager and predicate manager control the enablement and visibility of widgets based on the associated predicates and the current state of the application. Figure 4-1 shows the architecture of this system.

I will discuss his system only briefly, talking only about the more important things. Readers interested can find more information in the thesis [Gieskens, 1992], or, more briefly, in the paper [Gieskens and Foley, 1992].

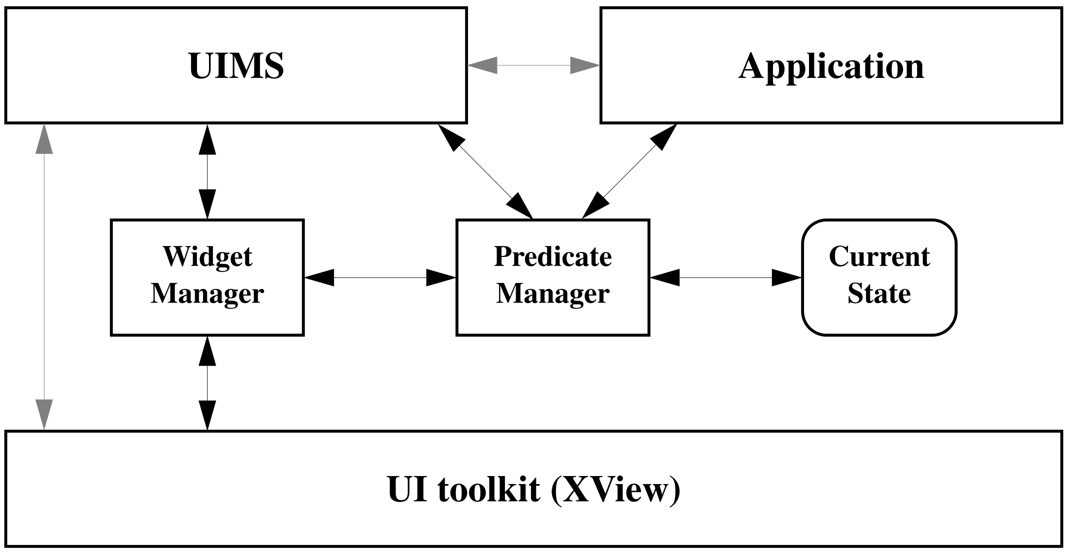

Figure 4-1
Architecture of the pre- and postcondition system

4.1.1 Architecture

The central part in the architecture is the `Current State`, which contains all the predicates that are true at this time. This defines the current state of the interface and application. To maintain consistency the `Current State` can only be changed by the `Predicate Manager`. The sole task of the `Predicate Manager` is to handle requests about the current state, and make changes in the current state. The application, `UIMS` and `Widget Manager` can all communicate with the `Predicate Manager`.

The `Widget Manager` controls all the widgets, and makes sure that the state of the widgets corresponds to the `Current State`. This means that the `Predicate Manager` tells the `Widget Manager` about any changes in the `Current State`. The `Widget Manager` can then update the widgets.

4.1.2 Object hierarchy

The interface system is implemented in C++, so all the entities in the system are objects. Instead of just listing all the objects I want to pick out one hierarchy , and tell something about it. I want to do this because most of the objects are very uninteresting, and much more detailed information about this can be found in the thesis, but this particular hierarchy has got some influence on the implementation of the help system.

The hierarchy I want to talk about is shown in Figure 4-2. It is the object hierarchy for the preconditions. This hierarchy is special because it shows how the problem of compound preconditions is solved. Compound preconditions are preconditions that contain more then one predicate, concatenated with AND and OR. On the highest level there is of course the `Precondition` object. This object represents the complete precondition, and is part of the `Widget` object, which I will not describe here. This `Precondition` object contains one or more `OR-expression` objects. Each of these objects represents a separate or-clause in the precondition, e.g. "or-expression-1 OR or-expression2". Each OR-expression consists of one or more `AND-expression` objects. Each of these objects represents a separate and-clause in the precondition, e.g. "and-expression-1 AND and-expression-2". Each AND-expression can also be negated, e.g. "and-expression-1 AND NOT and-expression-2".

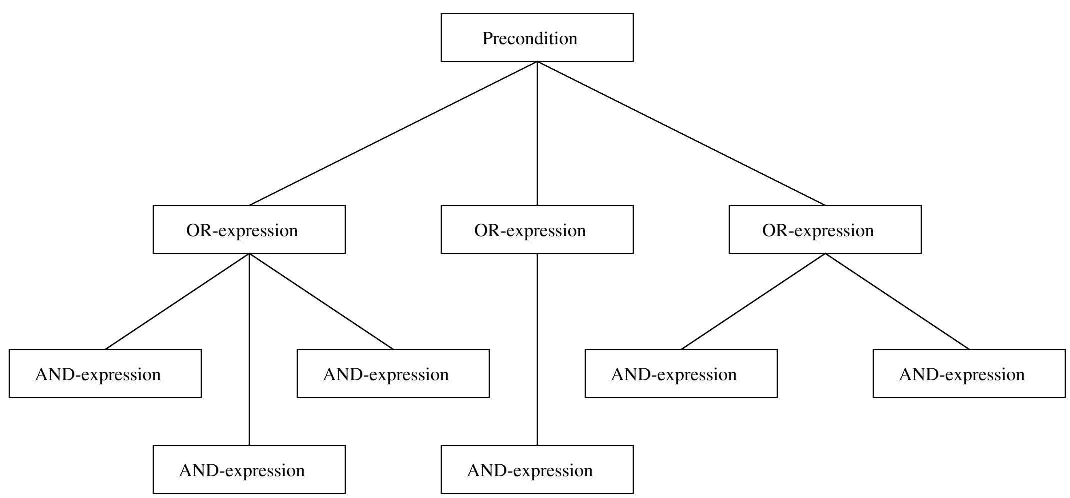

Figure 4-2
Object hierarchy for the preconditions

With this combination of AND, OR and NOT, all logical expressions can be made, but it might sometimes be hard to write and expression in this form. An advantage for the programmer is that this format is easier to parse, and it greatly simplifies the generation of help text, because each alternative precondition can be worked on in turn.

4.2 The help system

In this section I will describe the architecture of the help system. I will describe three parts of the help system. First I will describe the main architecture of the help system, which describes how the help system is notified of a help request, and how these requests are handled. Next I will describe the help-generating modules. First I will look at explaining why something is disabled, and after that I will look at explaining how something can be enabled. For the latter two section it is interesting to see how the templates and architecture map onto each other .

4.2.1 Help system architecture

Figure 4-3 gives an overview of the architecture of the help system. This figure shows the top-level of the help system architecture. The arrows denote the program flow. First, events are coming into the interpose function. This function receives all the events that the application receives. The `Interpose function` looks at each event in turn, and passes the events back to the system, unless it is a help event. The `Interpose function` passes help events to the help system, which can then generate help for it.

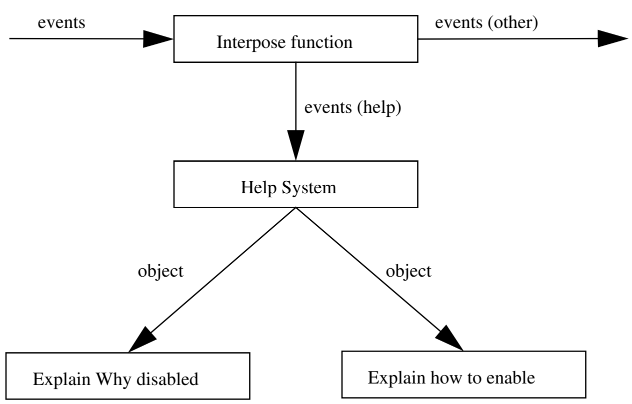

Figure 4-3
Architecture of the help system

The help events are passed to the `Help System`. The `Help System` will first try to match the help event to a registered widget, that is, a widget that has associated pre- and postconditions. If such a widget is found, then the appropriate explanation function is started, and the object is passed to it.

4.2.2 Explaining why

Figure 4-4 shows the architecture of explaining why something is disabled. The figure shows the template used to generate the text, with some part of it boxed. These boxed parts are the parts that will be generated. Each of these generated parts is a function that generates that part. The functions to determine the name and type of the widget are very simple. Since the widget itself is already given this is a simple matter of looking up the values. The `Reasons` are where the real generation takes place. The `Reasons` function is expanded in the figure.

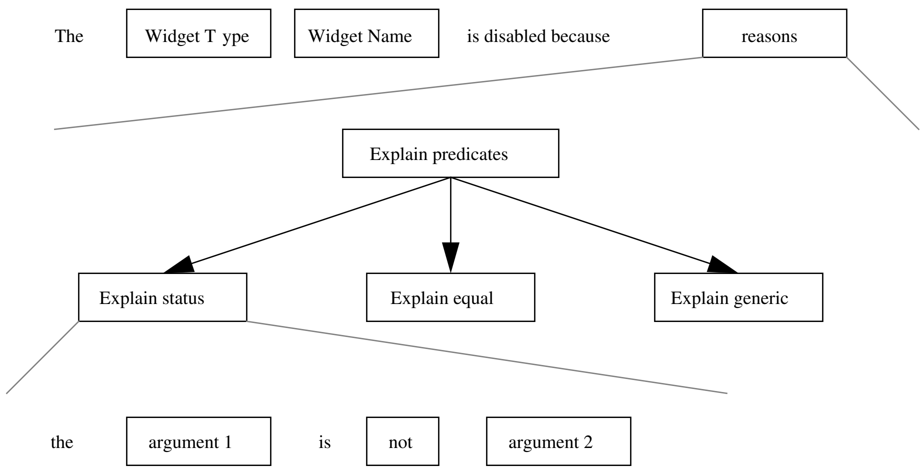

Figure 4-4
Architecture of explaining why

The reasons are generated by `Explain Predicates`. This function looks at each predicate in turn, and if the predicate is currently false it will be explained. `Explain Predicates` calls the appropriate function to generate text for the predicate, Each template has its own function, and there is also a generic function to handle all other predicates. As an example the template for the status() predicate is shown. This predicate will be explained in more detail in Section 4.2.4.

4.2.3 Explaining how

Figure 4-5 shows the architecture of explaining how a widget can be enabled. There are less templates here then with explaining why, because only the steps need to be displayed. This is just a list of interactions with the interface. This list is generated with a planning algorithm, in this case an adaption of the `STRIPS` algorithm. I already discussed this algorithm in Section 2.4.2 on page 17.

After all the steps have been generated they will be displayed on after another. All the information displayed here can be found easily in the information that results from the planning process. First the number of the step, followed by the action to be performed on the widget, followed by the widget type and name.

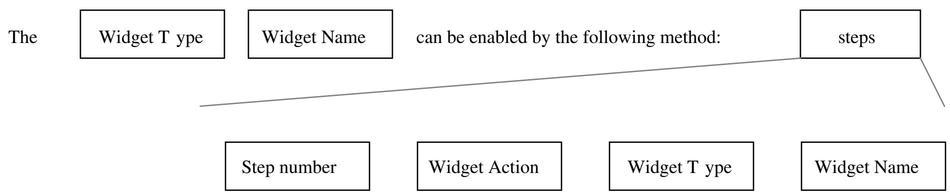

Figure 4-5
Architecture of How? help

4.2.4 Example: the status predicate

As an example of a predicate-specific routine I will describe the routine to convert a status predicate into text. This predicate is also visualized in Figure 4-4.

The status-predicate is a straightforward predicate. It is defined as status(object, status), e.g. status(Cannon, Loaded). A matching template is “The <object> is not <status>.”, in the example “The cannon is not loaded”. However, this template can be optimized. If the cannon could either be loaded or empty, then it would be better to say that the cannon is *empty*, instead of saying it is *not loaded*. Therefore the status procedure looks at the number of alternatives states for an object. If there are two states, it will drop the “not”, and use the other state, e.g. “The <object> is <other state>.” In the example: “The cannon is empty”.

4.3 The XView environment

The current implementation of the help system has been implemented in `Xview`, running in an environment of `OpenWindows 3.0`. In this section I will describe some problems and solutions that are typical for this particular implementation. The current implementation unfortunately is not very portable, because it uses the special help window facility which is imbedded in the `OpenWindows` window manager.

4.3.1 Help events

The help system has to know when the user wants to have help on a widget. There are several strategies for this. The best solution is probably to make the window manager to know about the help system, so that it can be called directly. This way the application need not even be aware of the help system, it would just be an integral part of the environment. However, to be able to do this the predicate system should be part of the window manager, and this was just not a feasible solution for us.

Because of these problems I decided to take a different approach. I decided to install an interpose function with each panel that was created within the interface system. This interpose function gets called first whenever events take place on the panel. This interpose function is shown in Figure 4-3 as the topmost function of the help system.

This approach works pretty good, but it has a couple of big drawbacks. For instance, it is not possible to detect whether somebody has asked help on a menu-item. Even though menu's can be part of a panel the panel doesn't receive any help-requests, so these help requests can not be sent to the help system.

4.3.2 Panels

Another very implementation-oriented problem occurred with the way panels and widgets are related in `XView`. In `XView`, all widgets are laid out on panels, and one panel can contain several widgets. Normally, this is used to group widgets together visually. The interface system also uses this hierarchical relation to group widgets, and if a panel gets disabled, the whole group of widgets on that panel get disabled. This layering is very nice, since the designer doesn't need to add a precondition to each widget, but he can just add it to the panel, and have it act on all widgets.

The problem with this is in generating explanations. Do I want to hide the existence of the panel to the user? The preconditions on the panel might be more general, and therefore more important. The current solution I have taken is to let the panels be transparent, that is, the preconditions of the panel are displayed as if they are preconditions of the widget.

4.4 Examples

To make everything a little bit more clear I have included some demonstrations in Appendix A of this thesis. Section A.1 contains the cannon demo. I have used this cannon throughout this thesis, but in this section I give the complete demonstration. This demonstration is very good at explaining most of the concepts of the help system, and is still small enough to be easily understood.

Section A.2 contains the VCR demonstration. This demonstration introduces one new concept which wasn't mentioned in the cannon demonstration, and it is also larger.

Finally , Section A.1 contains a demonstration of building an interface. In this demonstration a small interface is build from scratch.

Chapter 5

Conclusions

In this chapter I will draw some conclusions about the project, and I will talk about some of the things that can be done to extend this project.

5.1 Conclusions about the project

In Section 1.2 on page 3 I stated some questions that this research should address. Let's look at these questions again, and see what the results of the research are.

5.1.1 Creating help text automatically

I asked whether it would be possible to generate help text automatically . The answer is yes. Help text can be generated automatically . However , the designer still has to be aware of the fact that help text will be generated, and he should think about the generated help text while specifying the interface. Also, creating help text does not mean that the designer does not have to do anything about the help text. For instance, if an application contains very specific predicates the designer will have to provide specific templates for those predicates.

5.1.2 Quality of the help text

A more interesting question is the quality of the help text. Quality is very subjective, but I am interested in the possibility of generating text that is not obviously generated by a computer . The current template system works pretty good, although there are still some computer -like constructs. Also, templates are inflexible, and cannot always generate natural language help, like a human teacher would explain things. AI-researchers told me that the main focus in AI is in natural language recognition, and not in generating. They agreed that using templates is probably the most efficient solution. I don't see any major improvements here within the next few years.

5.1.3 Information needed

We need a logical representation of the interface in order to generate help text. In the current interface system this information is encoded with pre- and postconditions. This works very nice with the planning algorithm as well. We also need templates to create a framework for the textual help.

5.1.4 Conclusions

The research project has been successful. One of the goals was to see what could be achieved, given the current interface system. This research has shown that generating help text can be achieved based on this interface system.

5.2 Lessons learned

The lessons learned mainly concern setting up a project like this. I have learned two big lessons. One about building and designing large projects, and one about working in an interdisciplinary group of people.

5.2.1 The paradox of big projects

There is a paradox in building and designing big projects. On the one hand it takes so much time to understand an already existing project, understand its focus and its many aspects. On the other hand, big projects are much needed, because only then can be build on previous experience, and only then the benefits of a methodology can be really tested.

This is true also for the `UIDE` suite. It consist of so many tools that it is very easy to loose track of them. Understanding tools gets even more complicated because the main focus of `UIDE` only becomes clear gradually. Still, we learn a lot from `UIDE` because it is such a big project, en because so many people are working on it.

5.2.2 Multi-disciplinary work

Designing and building help systems is a very multi-disciplinary task. I was fortunate enough to do my research in a multi-disciplinary environment. Even though there was interaction between the different groups of people from Computer

Science, Psychology and other departments, I still found that a lot of people, including me, stayed within their own culture and group of people, instead of going out and finding experts on the other topics in your research. Fortunately, this was slowly changing, and more and more people from different departments started to work together, something I think is very important for the field of user interfaces.

5.3 Future challenges

I can see two future challenges. Adding animation to the help system, and providing adaptivity.

5.3.1 Mixing text and animation

As I have explained in Section 3.2.2 on page 25, text is very much suited for explaining abstract concepts. It fails, however, in explaining details on the screen, or explaining interaction techniques. Animated help has almost the opposite characteristics. It is very good in explaining interaction techniques, and pointing out details on the screen, like identifying an object. Animation is not very good at explaining abstract concepts, or explaining something at a higher level than interactions. It seems natural to bring these two together, since they complement each other so much.

As a matter of fact a previous project describing a system with only animated help [Sukaviriya, 1991] caused us to start this project with only textual help. We have now come to the conclusion that we should pursue research into a combination of textual and animated help. Some preliminary steps in this direction have already been made [Sukaviriya and de Graaff, 1992], but much work has to be done.

Combining textual and animated help can be realized with the current knowledge of the application, as described in Section 2.3.2 on page 12, Figure 2-2. The textual help will use all of the highest level, the application semantics, and parts of the middle level, the interface semantics. The animation part will use the lowest level, the interaction semantics, the compose the actual animation, and parts of the interface semantics to sequence the animation. In this way all the information in the model is used, as opposed to only textual or animated help, which uses just the top or bottom part.

5.3.2 Building adaptivity into the help system

Another important issue is adaptivity. As I described in Section 3.2.3 on page 25 the help system is context-sensitive. An adaptive help system is one step up from a context-sensitive one. Basically, an adaptive help system is identical to a context-sensitive one, but it can also adapt to the current user. We can still make use of all the techniques and knowledge used for the current system, but we need one more thing: a user model. We need to be able to model the user, and store information about her.

There is still much research to be done into adaptive interfaces and help systems, but maybe an example of its use will be useful. In Section 3.2.1 on page 23 I already pointed out problems with several alternatives for the same goal. The example was the pause button on a tape player, which could be enabled by both the play and record buttons. With an adaptive help system, information from the user model could be used to assess which of these two alternatives is more important to this user. Measurements done at all users could also be used to help novice users to learn the important, or at least often used, features.

References

Allen *et al.*, 1990 J. Allen, J. Hendler, and A. Tate, editors. *Readings in planning.* Representation and reasoning. Morgan Kaufmann Publishers, Inc., San Mateo, Calif., 1990.

Borenstein, 1985 N.S. Borenstein. *The design and evaluation of on-line help systems.* PhD thesis, Carnegie-Mellon University, 1985.

Card *et al.*, 1983 S.K. Card, T.P. Moran, and A. Newell. *The psychology of human-computer interaction.* Erlbaum, Hillsdale, 1983.

Chapman, 1985 David Chapman. Planning for conjunctive goals. Technical Report 83-85, Artificial Intelligence Laboratory, Cambridge, MA, 1985

Foley *et al.*, 1988 J. Foley, C. Gibbs, W. Kim, and S. Kovacevic. A knowledge-based user interface management system. In *Proceedings of ACM SIGCHI*, New York, 1988. ACM Press.

Foley *et al.*, 1989 J. Foley, W. Kim, S. Kovacevic, and K. Murray. Defining interfaces at a high level of abstraction. *IEEE Software,* 6(1):25-32, jan1989.

Foley *et al.*, 1990 J.D. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer graphics: Principles and practice*. Addison-Wesley, 1990.

Foley *et al*., 1991 — J.D. Foley, W.C. Kim, S. Kovacevic, and K. Murray. UIDE - An intelligent user interface design environment. In J. Sullivan and S. Tyler, editors, *Architectures for intelligent interfaces: Elements and Prototypes*, chapter 15, pages 339-384. Addison-Wesley, 1991.

Gieskens and Foley, 1992 — D.F. Gieskens and J.D. Foley. Controlling user interface objects through pre- and postconditions. In *Proceedings of the ACM SIGCHI*, Monterey, Calif., 1992. ACM Press.

Gieskens, 1992 — D.F. Gieskens. Controlling user interface objects through pre- and postconditions. Master's thesis, Delft University of Technology, apr 1992.

Heller, 1990 — D. Heller. *XView programming manual.* O'Reilly & Associates, Inc., oct 1990.

Kearsley, 1988 — G. Kearsley. *Online help systems.* Human/Computer Interaction. Ablex Publishing Co., Norwood, NJ, 1988.

Nilsson, 1980 — N.J. Nilsson. *Principles of artificial intelligence.* Morgan Kaufmann Publishers, Inc., San Mateo, Calif., 1980.

Norman, 1988 — D.A. Norman. *The psychology of everyday things.* Basic Books, New York, 1988.

Palmiter and Elkerton, 1991 — S. Palmiter and J. Elkerton. An evaluation of animated demonstrations for learning computer-based tasks. In Proceedings of the ACM SIGCHI, New Orleans, LA, 1991. ACM Press.

Sacerdoti, 1977 — E.D. Sacerdoti. *A structure for plans and behaviour*. Artificial intelligence series. American Elsevier, New York, 1977.

Sodoyer, 1990 — B.R. Sodoyer. *A standard help system with context sensitity*. Technical Report 90-33. Department of Information Systems, Delft University of Technology.

Sukaviriya and Foley, 1990 — P. Sukaviriya and J.D. Foley. Coupling a ui-framework with automatic generation of context-sensitive animated help. In Proceedings of the ACM SIGGRAPH symposium on User Interface Software and Technology. ACM Press, oct 1990.

Sukaviriya, 1991 — P. Sukaviriya. *Automatic generation of context-sensitive animated help.* PhD thesis, George Washington University, 1991.

Sukaviriya and de Graaff, 1992 — P. Sukaviriya and Johannes J. de Graaff. *Automatic generation of context-sensitive "show & tell" help.* Technical report. Georgia Institute of Technology.

Sun, 1990 — Sun Microsystems, Inc., Open Windows Developer's Guide V1.1, Reference Manual. Revision A, June 1990.

Szekely *et al*., 1992 — P. Szekely, P. Luo, and R. Neches. Facilitating the exploration of interface design alternatives: the Humanoid model of interface design. In *Proceedings of the ACM SIGCHI*, Monterey, Calif., 1992. ACM Press.

Wiecha *et al.*, 1990 — C. Wiecha, W. Bennett, S. Boies, J. Gould, and S. Greene. ITS: A tool for rapidly developing interactive applications. *ACM Transactions on Information Systems*, 8(3):204-236, jul 1990.

Wilenskey, 1984 — R. Wilensky. Talking to Unix in English: An overview of an On-line Unix Consultant. *AI Magazine* 5(1):29-39, Spring, 1984.

Wilkins, 1988 — D.E. Wilkins. *Practical Planning.* Representation and reasoning. Morgan Kaufmann Publishers, Inc., San Mateo, Calif., 1988.

Appendix A

Demonstrations

This Appendix contains three different demonstrations. Normally , I give these demonstrations on-line during presentations, or while demonstrating the system to visitors. These demonstrations almost always convey the important aspects of the help system. I have tried to give these demonstrations here as well, combining screenshots with text.

A.1 Cannon demo

This section contains the cannon demo. W e have already seen this demo throughout the thesis, but now I will give a complete demo with this sample program. In

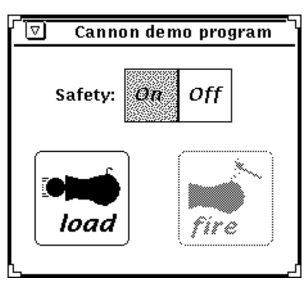

Figure A-1
Sample interface

Figure A-1 the interface is shown again. The program simulates a simple cannon, which also contains a safety switch. The safety switch can either be on or off. If the safety is on the cannon can not be fired. The load button loads the cannon, which can only contain one bullet. The fire button shoots the cannon. It should be obvious that the cannon can only be fired if it is loaded, and if the safety is off. The cannon can only be loaded if it is empty . In the figure the fire button is disabled, and the safety is turned on.

Let’s look at the help that gets generated for this interface. The fire button is disabled, and Figure A-2 shows the help window for this situation. It explains why the fire button is disabled, and how it can be enabled.

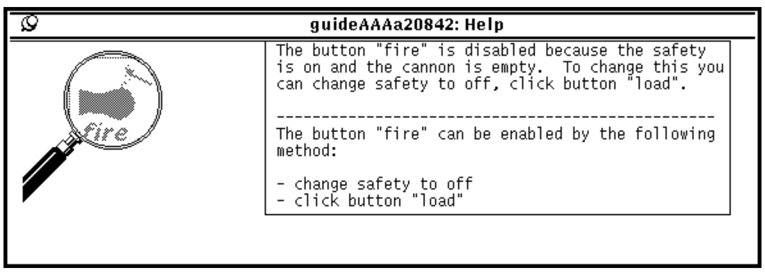

Figure A-2

Help window for fire button

As we can see in Figure A-2 there are two reasons for the fire button to be disabled, the cannon is not loaded, and the safety is on. If we change one of these precondition by loading the cannon, then the help text changes. This new help text is shown in Figure A-3. The cannon is now loaded, so the help system only talks about the safety .

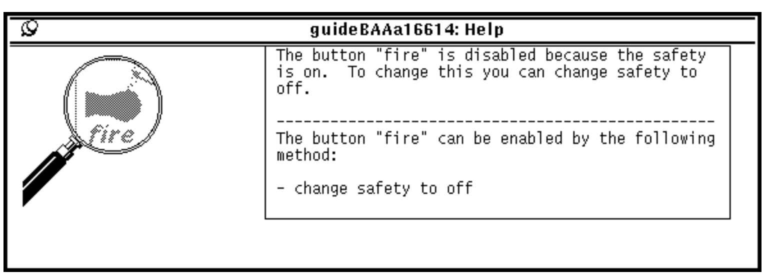

Figure A-3

Help window for fire button, cannon loaded

By now things have gotten more complicated if we look at the load button. The load button is now disabled because the cannon is already loaded. Obviously we can solve this by clicking the fire button. However , this button is disabled also. It will be clear that we first need to change the safety to off, before we can click the other buttons. Figure A-4 shows the help window for the load button in this situation.

Figure A-4
Help window for load button

A.2 VCR demo

The VCR demo is a bit larger than the cannon demo from Appendix A.1. First I will show some of the same kinds of help that we already saw with the cannon demo, although they will be larger here, and in the end I will show one new concept which isn't included in the cannon demo.

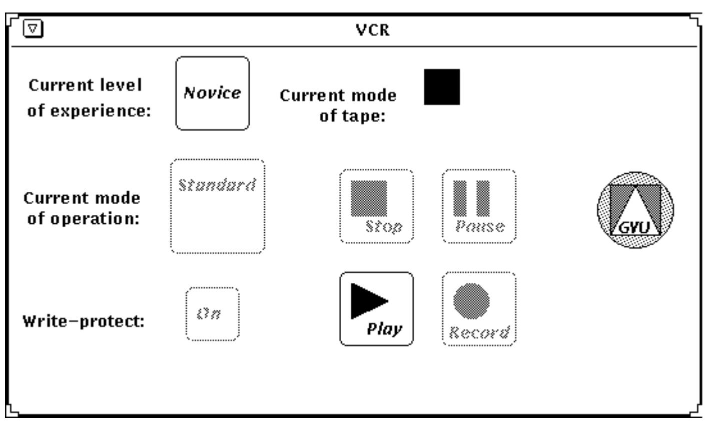

Figure A-5
VCR demo

Figure A-5 shows the VCR demo. This application simulates a simple VCR. It has the standard buttons for recording, playing, stop and pause. In addition there is a button which selects the current user mode. This mode can be either novice or expert, and controls the availability of certain features. More advanced features like recording are only available to the expert. There is also a button to toggle the write-protect of a tape, and a button to turn on programming mode. We will not use programing mode in this demo, though.

Let's say we want to record something. The record button is disabled, so we ask help about it. This help is shown in Figure A-6. It explains that we first need to change the experience level, after which we need to change the write-protect status. Then we can use the record button.

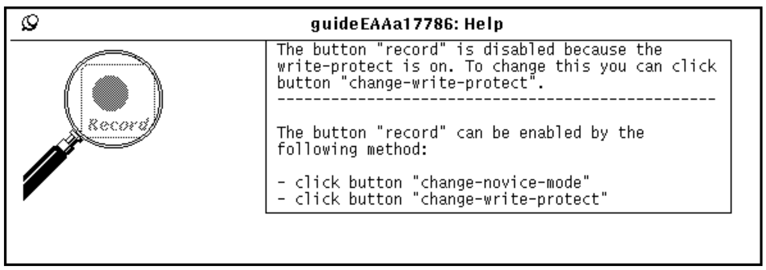

Figure A-6

Help window for record button

There is something interesting about the pause button in this demo. It can be enabled in two different ways. Obviously, the pause button can be enabled by clicking the play button, but it can also be enabled by clicking the record button. How the help system handles different alternatives is shown in Figure A-7 and Figure 1-8. The first figure explains why the pause button is disabled, and what can

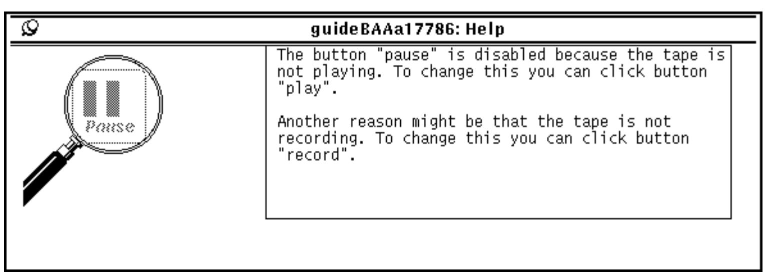

Figure A-7

Help window for pause button

be done about it. It first explains the play part, assuming this is the most important or most used part. Then it displays the record part as an alternative. This order can be chosen by the designer. The second figure explains how the pause button can be enabled. Here it presents the two alternative methods in the same order , and the record part includes the other buttons that have to be pressed before the record button can be used.

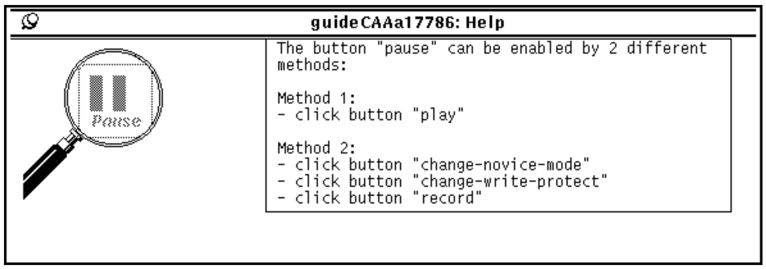

Figure 1-8

A.3 Building a simple interface

In this demonstration I will build a simple interface. I will use a special version of DevGuide, an interface builder from Sun Microsystems. At the GVU center we have extended this interface builder to include the pre- and postconditions, and also the interface and help system.

Our goal is to make an interface like the one in Figure A-9. This interface contains two buttons labeled “on” and “off”, and they will turn a fictious machine on and off. These buttons are mutually exclusive, the on button can only be used if the machine is turned off, and vice versa.

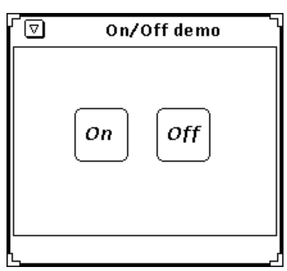

Figure A-9
A simple interface

First we generate the interface itself. I will not describe this part here, because this is all standard DevGuide, and not very interesting for this demonstration. After this stage we have an interface that looks exactly like the one in Figure A-9. However , it hasn’ t got any functionality . We will add this functionality in the following steps, using the interface system.

First we will add some functionality to the “On” button. This button can only be enabled if the machine is off, so we add a precondition “status(Machine,Off)” to its enable-precondition. Then, if this button is invoked, “status(Machine,Off)” has to be removed from the current state, and replaced by “status(Machine,On)”. A part of the property-sheet for the “On” button is shown in Figure A-10. This figure shows the part that is used to enter the pre- and postconditions. There are preconditions for controlling visibility and enablement, and postconditions that will add and delete predicates from the current state when the button is clicked.

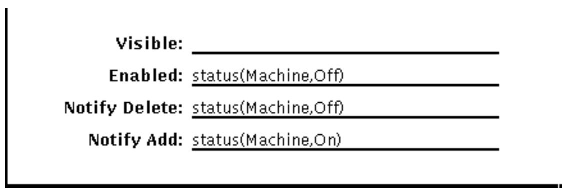

Figure A-10

Property-sheet for “On” button

We will now do identical steps for the “Off” button, but with all the predicates reversed. Again part of the property-sheet for this button is shown in Figure A-11.

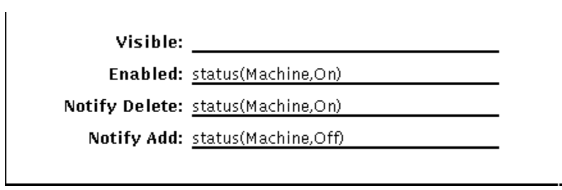

Figure A-11

Property-sheet for “Off” button

We have now defined the functionality of the interface. Depending on the status of the machine one of the two buttons is enabled, and selecting it results in changing the status of the machine, so that the other button becomes enabled. There is one additional thing to specify , the initialization. The interface system needs to know in what state the machine is when we start the application. This is specified on the property-sheet of the main window of the application. Part of this property-sheet is shown in Figure A-12.

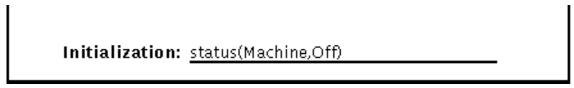

Figure A-12
Initialization

Now we can interact with the interface. On startup the machine is off, and the “On” button is enabled. We can now switch the machine on, and back off again. Since all this functionality was achieved by the interface system we also have the help system available to give help about this application. Figure A-13 shows the help window for the “Off” button.

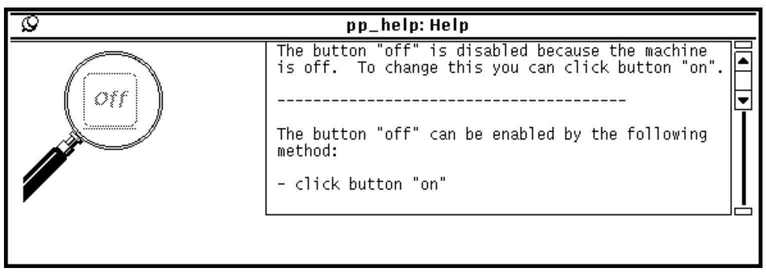

Figure A-13
Help window for “On” button