A Visual Method for Input of Uncertain Time-Oriented Data

by

Manish Goyal

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1997

Author . . . . . .
Department of Electrical Engineering and Computer Science
August 29, 1997

Certified by . . . . . .
Peter Szolovits
Professor
Thesis Supervisor

Accepted by . . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

A Visual Method for Input of Uncertain Time-Oriented Data

by

Manish Goyal

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 1997, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

An expert system that reasons with temporal knowledge must obtain that temporal data before being able to process it. In the case where a human is required to provide this information, an effective means of data entry must be established. The task for the human becomes more complicated when the system also requires knowledge of "temporal uncertainty" (the notion that certain events took place somewhere in the span of a time interval rather than at an exact moment in time).
This paper describes the design and implementation of the GA Clock Interface, which was created to simplify the task of representing temporal uncertainty when entering time-dependent data into a computer. The user interface paradigm known as direct manipulation plays a significant role in the design of this UI. The need for such an interface is discussed, followed by a detailed description of its implementation.

Thesis Supervisor: Peter Szolovits
Title: Professor, Medical Decision Making Group

Acknowledgments

I would like to express my thanks to Sean (mesozoic) Doyle, who guided me in my design and implementation. This project would have been nightmarish without his advice.

I would also like to thank Prof. Szolovits for supervising this project. I undoubtedly tested his patience, but he persisted in demanding from me a project of which I could be proud.

There is no way for me to articulate the depth of encouragement and support I received from Charu Puri and Hemant Taneja. They sacrificed countless hours to assist me as I approached the final days of this thesis. I owe them big.

And finally, I would like to thank, with love and affection, my mom, my dad, and my two brothers, Ashish and Amit Goyal. What of MIT could be worth my persistence and perseverance, without their love and support?

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Temporal data refers to any data that is time-dependent. Such data is often intrinsically associated with a term called temporal uncertainty, which refers to the notion that there exists a region of time (a time interval) during which a particular event is certain to have taken place. An effective means of collecting and representing temporal uncertainty becomes necessary when large amounts of temporal data need to be analyzed. This thesis describes the development of the Guardian Angel Clock Interface (GA Clock), which is a direct manipulation interface system that accounts for temporal uncertainty in the collection of patient data.

The extent to which an expert system should be able to handle temporal data and, consequently, temporal uncertainty, has been a topic of ongoing research. Traditionally, Artificial Intelligence (AI) programs have ignored the role of time-dependent variations in data when formulating hypotheses and conclusions. These programs have typically taken the one-shot approach, where all the information used in analysis is available at a single consultation and a conclusion is drawn therein [14]. Such an approach can lead to inadequate analysis and results, as some expert systems are used in the management of ongoing, time-dependent processes. This holds especially true in the case of medical expert systems, where data is inherently time-dependent in the onset, development, and treatment of disease. In 1987, Isaac S. Kohane [9] wrote,

> *Expert system performance and knowledge acquisition is at best incomplete and at worst fatally flawed if the system does not possess a systematic and principled "understanding" of temporal knowledge. Therefore, even though temporal reasoning is not in and of itself suffi-*

cient to produce satisfactory behavior in expert systems, it is a necessary and significant step in that direction.

Kohane's doctoral thesis was an investigation into the effect that a systematic representation of temporal relations can have on the process of diagnosing disease. As part of this investigation he developed the Temporal Utility Package (TUP) to represent qualitative and quantitative temporal relations along with a variety of temporal contexts. He also developed Temporal Hypothesis Reasoning In Patient History-Taking (THRIPHT), a prototype expert system that demonstrated "TUP's application and the role of temporal reasoning in different phases of the diagnostic process: data gathering, hypothesis evocation, elaboration, instantiation, and hypothesis ranking."

In reasoning with temporal uncertainty, TUP and THRIPHT use special forms of backtracking for representation and analysis; however, its capabilities are limited. Kohane explains some of the challenges involved with sophisticated reasoning using temporal uncertainty, noting that it is often difficult to obtain time-varying clinical data with quality probabilistic distributions. Of course, an effective data entry system for statistics and their uncertainties would make such data more easily obtainable. Kohane goes on to explain that an expert system which is capable of sophisticated temporal reasoning could be a worthy ally to the doctor in any diagnostic process, since the use of temporal data (namely, the patient history) is "the most powerful and widely used diagnostic tool of health-care practitioners."

Further research into reasoning and analysis using time dependent data was conducted by Thomas Anton Russ, who developed a formalism called the Temporal Control Structure (TCS). Russ's goal was to simplify the construction of knowledge-based systems that perform monitoring and management of complex time dependent processes. TCS is a model for temporal reasoning which performs the necessary bookkeeping to manage data

dependencies over time. As part of his work Russ developed several reasoning systems using TCS, the most ambitious of which was able to track the progress of patients suffering from the disease ketoacidosis over the course of several days.

The impetus for Russ' work was the "need to improve the technology for designing intelligent monitoring systems." At the heart of any monitoring and management system lies the ability to accept and react to changing data. The fundamental idea exploited by Russ is that *the construction of time-dependent expert systems can be simplified by dividing reasoning into static and dynamic components.* Identifying dynamic components, those that rely on time-dependent variables, limits the work performed by an expert system to that which is absolutely necessary when temporal data changes. As a computational model for time based reasoning, TCS isolates the effects of temporal change in a given process and simplifies the task of the expert system. This facilitates "the efficient updating of the state of knowledge of the system."

Kohane and Russ are only two members of a large community of researches that are still actively seeking to formalize the way temporal data and its accompanying uncertainty are used in expert system analysis. Whether systems are being built for clinical diagnosis, the control of large scale manufacturing processes, or the tracking of information flow within an organization, effective handling of temporal data is a necessary step towards optimization. The process of data collection cannot be ignored in this effort.

1.2 Motivation

The user-interface which was developed in this thesis will from here on be referred to as the Guardian Angel (GA) Clock Interface. As part of the ongoing Guardian Angel project for developing Patient-Centered Health Information Systems, the GA Clock strives to meet a portion of the goals outlined in The Guardian Angel Manifesto [18]. The Mani-

festo articulates the hope to dramatically improve health maintenance and health care delivery "by making useful information available in a timely manner to the patient" and by investigating 'many of the technical issues of how to collect, maintain and interpret comprehensive health information to speed the advance of medical informatics.'

The GA Clock is designed to facilitate data entry when a person is required to input temporal data into the system. The remainder of the system can consist of data analysis conducted by an automated process or a human. A specific example of the utility of the GA Clock is evident when considering the case of a diabetic patient. Such a patient may need to inform his doctor of the times he eats his meals, takes his insulin, and checks his blood sugar level in the course of each day. This information can assist the doctor in making recommendations for dietary changes or in adjusting the prescribed dosage of insulin to be taken. Ideally the patient would keep a precise record as follows:

> Finished breakfast at precisely 8:30 am
> Checked blood sugar at precisely 10:53 am
> Injected insulin at precisely 11:23 am
> etc....

Such a record of each event as it has just taken place would of course provide the most accurate data. It is unreasonable to expect, however, that any person will (or even can) diligently maintain such a record. The patient would be more likely to record the day's events at the end of each day, inadvertently introducing temporal uncertainty in the time at which these events occurred. Even so, it can still be valuable for the doctor to know the time intervals during which the patient believes the above actions took place. The following information can be enough to perform an analysis upon which the physician can base his recommendations:

> Ate Breakfast at around 8:15
> Checked blood sugar at approximately 10:45, could have been

as late as 11:00, but not before 10:40

Injected insulin at around 11:20, give or take 5 minutes

etc....

Direct input of such data into a computer by patients is more efficient for transmitting and processing than writing it on a paper form. Thus, the goal becomes designing a method of data entry which requires minimum effort. Several options exist for this type of data entry. A forms-based application could be used by the patient to manually enter each activity and its corresponding time of occurrence (including the interval of uncertainty). However, this requires the user to conduct six separate actions to convey a single activity. For example, to record the fact that an insulin injection was taken between noon and one p.m., the user must enter the fact that an injection was taken, enter the start and end times of the period of uncertainty, and tab three times to reach each field. Each entry requires multiple keystrokes and every additional keystroke increases the probability of error by the user. Data entry via such a form would prove tedious and may deter even the most determined user. Almost as difficult is the task for the physician who must read the data on this type of form. Since the information is provided purely in text, no graphical information is conveyed about the actual event being recorded or the duration of the intervals of uncertainty.

A menu driven pull-down list offers a simpler form of data entry but still requires at least three operations to record an event. Using the above example, the user must select "Injection" from a list of available options, then select and enter values for each of the "start-time" and "end-time" fields.

Several limitations are apparent upon examination of the above data entry methods. The number of operations required of the user is high if he is expected to record every event from every single day. User feedback is almost completely lacking in that the inter-

faces do not present a graphical representation of the data that has been entered. Thus, if the patient himself is to draw any conclusions based on the data, or even if he wishes to simply verify that he has remembered to include all the activities, he must backtrack through the entire forms based application and check each field, or re-select every option on the menu, to confirm that all the data was appropriately entered. As would be the case for the physician who has to examine these forms, the patient must draw conclusions via mental calculations of the data as seen in text rather than from graphic representations of the times and associated uncertainties of each event. Until recent times such data entry limitations have hindered the progress of medical informatics for non-intensive management of patient treatment.

The Guardian Angel Manifesto provides the example of how medical informatics could assist in the treatment of patients with diabetes mellitus. It is stated that greater patient education, increased clinician/patient communication, and heightened responsiveness to changes in physiology or life-style could meet the requirements for more stringent glycemic control and thus reduce the risk of progressive chronic complications. However, attempts to provide these patients with automated assistance in the management of their blood glucose control over the past ten years have not met with lasting success for a variety of reasons, one of which clearly has been inadequate user interfaces.

The GA Clock is designed to specifically address this aspect of medical informatics. Whether data with temporal uncertainty is to be used by an expert system which analyzes and forms diagnostic hypothesis, or by a physician who formulates a diagnosis himself based in some part on this data, it is a necessary task that the data must be effectively collected and communicated in a reasonably simple manner. As part of the overall Guardian Angel system, the GA Clock can make progress towards meeting the following two objec-

tives, which are a portion of the primary objectives outlined in the Manifesto for patients with Diabetes Mellitus:

(1) Decreased patient reliance on expensive tertiary care specialists for routine advice on home management of diabetes mellitus along with increased timely communication of relevant clinical data to the diabetologist (nurse or physician).

(2) Improved clinical data collection to help conduct outcomes research and suggest improved therapeutic interventions.

Improved data collection and timely communication of clinical data go hand in hand; the two are inseparable. Various UI principles were explored and used to meet the objective for efficient data entry via the GA Clock Interface. These principles and their implementation are described in the chapters that follow.

Chapter 2

Closed Intervals: A Representation for Temporal Knowledge

2.1 Introduction

The analysis of time-varying clinical data is an essential part of therapeutic decision making. Advances in instrumentation and database technology have provided the means for collecting, storing, and retrieving large amounts of this data. As this technology has improved, the amount of data collected and stored has increased to a point where determining the clinical implications of this data has become a complex task. Appropriate representation of the data for effective diagnosis has been the focus of research in the medical informatics community for many years [6]. One of the key challenges in this task has been establishing an effective representation that accounts for the temporal uncertainty associated with clinical data.

It is the assertion in this thesis that in the absence of certain knowledge of time, a useful representation scheme could use uncertainty bounds. Essentially, events of interest which are often thought of as occurring at precise moments in time (e.g. "I took the shot at 1 p.m.) are actually time estimates with a bounded uncertainty. Allen (1983) proposes that time intervals can be used to represent this uncertainty [1]. These time intervals can then be used to form relationships with each other, and to draw conclusions based on those relationships.

Time intervals in this sense have been used to perform constraint based reasoning in AI systems [21]. The general idea has been to use existing information about the relations among time intervals to reach conclusions about other intervals. For example, if interval

A occurs before interval B, and interval B occurs before interval C, then interval A necessarily occurs before interval C.

This chapter begins with a description of temporal uncertainty, followed by a review of classic data representation schemes. The pros and cons of interval versus point representations are discussed. Following that is a description of how to incorporate uncertainty into these representations and a brief discussion of uniform versus non-uniform uncertainty.

2.2 Uncertainty of Temporal Data

Temporal data refers to any data that is time-dependent. Examples of such data include records of the exact moments in time when certain events took place, as well as the values assumed by a variable which changes over time. Temporal uncertainty refers to the notion that there exists a region of time during which a particular event is certain to have taken place. Data with temporal uncertainty does not pinpoint the exact moment an event has occurred, but rather a time interval during which that event is guaranteed to have occurred. Temporal uncertainty allows for a less restricted interpretation of data in that exact moments of time are not specified. In the observation of real world events, where adequate sensing and recording devices cannot always be implemented, it is often impossible to obtain temporal data that does not contain elements of uncertainty. When large amounts of this data need to be collected and analyzed, effective means of entering and representing temporal uncertainty become necessary for any system or human that wishes to use that data.

An example of clinically relevant temporal data is a time history of a patient's blood pressure measurements. The precision of the times at which the measurements were taken is strictly dependent upon the data acquisition process. For instance, if the process of data

measurement is coupled with a computerized data acquisition system, the temporal uncertainty of the measurements are minimal. However, if a nurse is taking these measurements then there exists an inherent delay in the actual time of the event and when it is recorded. This delay, along with the probability of human error, make it likely that the recorded time will contain elements of uncertainty. Another similar example is a coroner's estimation of the time of death of a heart attack victim (e.g., time of death recorded as "between 3:15pm and 3:30pm"). In this case a substantial delay between the occurrence of the event and the time at which it is recorded is likely to exist, thus introducing a relatively large interval of uncertainty.

The above examples demonstrate the uncertainty in temporal data which arises due to errors in pinpointing the exact time of occurrence for a particular event. However, when recording the uncertainty for a process which is sustained over a period of time, determining the uncertainty for the duration of that event requires additional analysis. For example, if a patient injects his medication at approximately 12 p.m. and the medication's effective duration is to be estimated, it is necessary to know (1) the uncertainty surrounding the actual time of the injection and (2) the estimate of the duration of the medication's

effectiveness along with the uncertainty bounds around that estimate. The total estimation accounting for both uncertainties can be calculated pictorially as indicated in Figure 2.1.

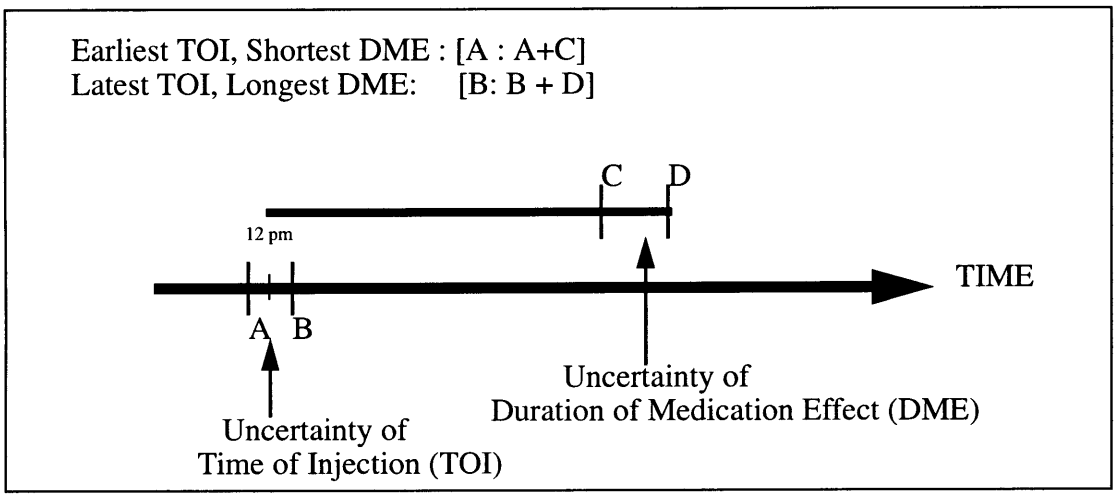

Figure 2.1: Various uncertainties in temporal data

This example uses hard bounds on the intervals of both uncertainties to calculate a final estimation of pharmacological duration. The next section begins a discussion on the existing methodologies for trying to represent different kinds of temporal data.

2.3 Temporal Representation Schemes

Representation of temporal data is a key determinant for the types of conclusions that can be drawn from analyzing that data. An ineffective methodology can misrepresent the temporal relationships that exist among time-oriented data and thus result in the loss of valuable information necessary to make accurate diagnoses. Subsection 2.3.1 describes the *point event* and *interval event* representation schemes. Subsection 2.3.2 discusses how to incorporate uncertainty into these schemes by using an abstract concept called a *time line*.

2.3.1 Point Event and Interval Event Representation

Point events describe events that have no meaningful duration and are atomic, meaning that they are all described by the same, smallest possible unit of duration. Hence, these events can only occur before, after, or at the same time as other point events. For example,

an injection can be described as a point event since the actual time to inject is minimal and of little consequence. In general, the process of taking patient readings can also be described as point events. Traditional medical databases store data as point events using what is known as the time-oriented data (TOD) model [20]. Storing information in these databases as <attribute, time, value> tuples makes for simple and flexible data entry and analysis. [6]

While this type of representation may be commonly used to record large amounts of event driven patient data, these point events *by themselves* do not allow for the expression of uncertainty around a central time. For example, additional information is needed to express that an injection was given between 9:55 a.m. and 10:05 a.m., with the likely time of injection occurring at 10:00 a.m. In the absence of exact knowledge for the time of occurrence an event, a mechanism for representing uncertainty must be devised to perform accurate analysis.

Furthermore, point events are inherently lacking when it comes to representing processes which are sustained over time. For instance, the point-based representation may be enough to record the time at which a patient's medication was given, but it cannot be used to convey the observed duration of the medication's effect (as in the case of a patient given aspirin at 12 a.m., whose fever was reduced for 6 hours). Since determining the clinical implications of patient measurements may require knowledge of processes which are not point-based, point event representation can cause problems in analysis. For example, a patient's blood pressure measurements have a different meaning if it is known that the patient was suffering from a heart attack at the time of the examination. The presence of a heart attack, the duration of a drug's effect, and the presence of a fever are examples of information that do not fit the point-based model.

Cousins et. al. state that an *interval event* can be used to represent any process, state, or event that has a meaningful duration associated with it. As mentioned above, the effective presence of a drug, the time length of a fever, and the duration of a heart attack do not fit a point-based model. However, these can all be effectively described via interval event representation, where the interval corresponds to the duration of the process or event. Unlike point events, these events are not atomic in that they can contain or overlap with other interval events. Like point events, interval events also do not intrinsically express uncertainty around the duration of a given process. The next subsection discusses how Cousins et. al. allow for representations of uncertainty around both point events and interval events in a temporal information system called a time line.

2.3.2 The Time Line Abstraction

Representation of the temporal data is done on an abstract entity called the *time line*. In this temporal information system, point events and interval events can be represented as sequences of chronological events. Interval events are represented by horizontal boxes whose lengths correspond to the duration of their intervals. Point events are placed appropriately according to the event's time of occurrence. Naturally, these point events can be contained within other interval events.

Uncertainty is represented for both of these events using line segments to denote the range of uncertainty. The interval of the line segment is always larger than the interval of the event being represented. For example, a line segment can be drawn in the time line to indicate some range of time. Then, by placing a horizontal box inside that line segment (the length of the horizontal box is necessarily shorter than that of the line segment), an interval event can be specified to have occurred at any time in the larger interval denoted by the line segment. Similarly, a line segment can be used to indicate the range of uncertainty around a point event. Hence, Cousins et. al. use the notion of a bounded interval

(denoted by the line segment) to represent uncertainty around a point event or interval event.

By supporting both the point-based model and interval-based temporal data, the time line is a more complete abstraction. Sets of time lines can be displayed and manipulated for collective analysis in an interactive environment called the *time line browser.*

The unit length of duration used in a given time line depends on the granularity one wishes to represent, which is in turn dependent upon the context of the situation being described. For example, a time line which is used to record events taking place within an intensive care unit (ICU) will likely be described with a high temporal resolution; the unit duration in this context could be seconds since urgency demands that events are monitored and recorded very quickly. However the events recorded to monitor a disease that spans the course of several months is likely to be described at a much lower resolution; the unit duration in this case could be days or even weeks. If the two time lines which are so characteristically different are overlayed to make any useful clinical observations, a loss of information could occur for the data gathered at the higher resolution. The entire ICU time line could effectively be reduced to a single interval or even a single point event (which may or may not be desirable) depending on its scale relative to the lower granularity time line.

2.3.3 Uniform Interval Uncertainty

A point event has a uniform uncertainty associated with its occurrence if it could have occurred anywhere in a given interval with equal probability. For instance, a diabetes patient could recall that he administered an insulin shot at some time between 11:10 am and 11:20 am (Figure 2.2). In such a case, the interval representation should be able to

successfully convey that the shot could have been taken at any time from 11:10 am to 11:20 am with equal probability.

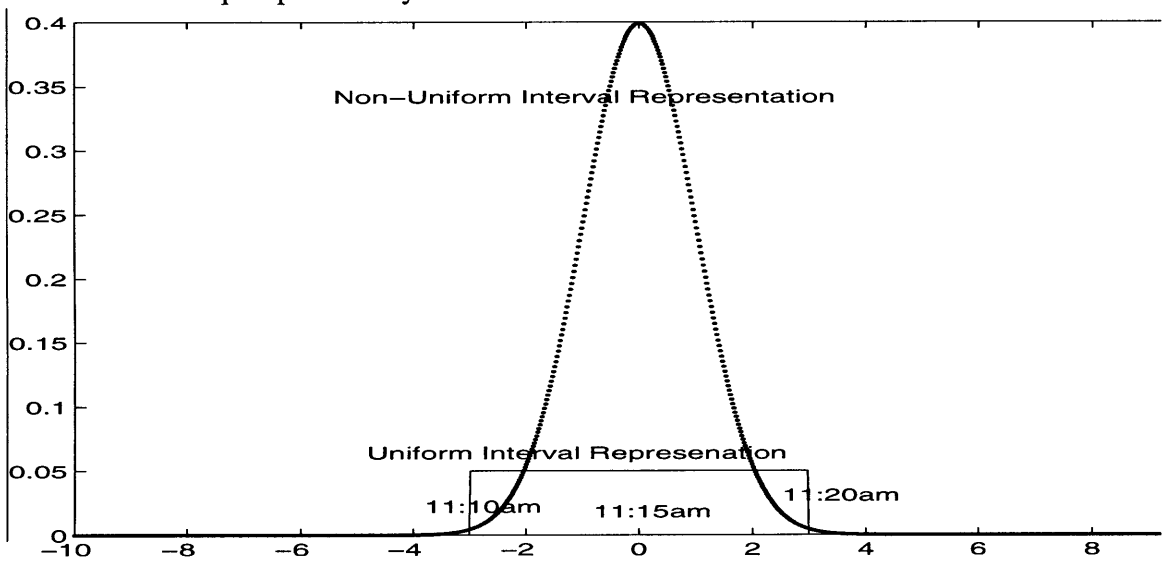

Figure 2.2: Uniform versus Non-uniform uncertainty regarding the occurrence of an event.

2.3.4 Non-uniform Interval Uncertainty

An event has a non-uniform uncertainty associated with it if it could have occurred at anytime within the given interval but most probably occurred at a specified time within that interval (also shown in Figure 2.2). For instance, the diabetes patient from the previous example might also want to convey that the insulin shot was most probably taken at 11:15 am. In order to capture this additional information, the probability of 'when the shot was taken' could be approximated using a normal distribution centered around 11:15 am.

When using temporal data for analysis, it is important that both uniform and non-uniform uncertainties are capable of being represented adequately.

2.4 Conclusion

This chapter describes the key characteristics of temporal data. The usefulness of point and interval representations in capturing these characteristics are discussed. All events have an inherent uncertainty associated with their occurrence. This uncertainty can be

classified as uniform or non-uniform uncertainty. The events which have a significant duration (interval events) have an added uncertainty regarding the length of their duration. All these uncertainties need to be effectively captured in a visual representation scheme. It is proposed that bounded uncertainty intervals are a useful means for representing point events with uncertainty and drawing conclusions based on this temporal data.

Chapter 3

Principles of User Interface Design

Any system which cannot be well taught to a layman in ten minutes,
by a tutor in the presence of a responding setup, is too complicated.
Nelson, Ten Minute Rule, 1974 [13].

3.1 Introduction

If a system is to require any level of human-computer interaction, the *user interface* (UI) can be the single most important factor in determining the success or failure of that system. Consequently, enormous monetary and human resources are spent in the design and construction of effective UIs. Sutton and Sprague (1978) found that, on average, the user interface comprised 59 percent of the total code in a given application, sometimes comprising as much as 88 percent [17]. In AI knowledge-based systems it was found that UIs constitute between 40 to 50 percent of the overall code [4].

There is a strong need for easy to use interfaces in medical systems since timely and accurate communication of patient data is essential for proper analysis and diagnosis. This holds especially true if the primary user of the system is a novice to computers, as could easily be the case for many of the applications developed as part of the Guardian Angel project. The remainder of this chapter discusses the principles of user interface design that were employed in the construction of the GA Clock Interface. Section 3.2 describes the concepts used by some experts to qualitatively assess and evaluate human-computer interaction. Section 3.3 defines the term *direct manipulation* and discusses the advantages of interfaces built with this particular style. The final section describes several attributes that should be present in a friendly and easy to use UI, as articulated in the Macintosh Human Interface Guidelines [2].

3.2 Minimization of Cognitive Resources

From the days when programmers used punch cards to instruct computers to perform simple arithmetic, to the advent of video games in which users manipulate complex graphical objects by moving a joystick and pushing a button, human-computer interfaces have ranged from mundane and cumbersome to inviting and remarkably easy. It is the hypothesis of many experts, and the central idea on which the work in this thesis is based, that people are more capable of performing (or at least more willing to perform) a small number of complex operations than a large number of simple ones to accomplish a given task. This section supports this argument by discussing the concepts in human-computer interaction developed by Hutchins, Hollan, and Norman (1986) to address this very idea [7].

Hutchins et. al. assert that the ease of a user interface rests partially in the amount of cognitive resources a user must commit to perform necessary actions. The term *distance* is used to describe the gulf that exists between a user's goals and the way his goals must be specified to the system. For instance, many modern windows based operating systems give users the ability to display the contents of a file by directing the mouse pointer to the appropriate file icon and double-clicking over that icon. In this example the user's goal is accomplished by a complex action that involves directing the mouse pointer appropriately and clicking a button. Certain cognitive resources are demanded of the user to carry out that action, thus introducing a distance to reach his goal. Consider an example of the same goal being sought but with the mouse no longer functioning. Now the user must initiate a key sequence to select the appropriate window which contains the icon, then another key sequence to select the appropriate file icon, and finally some key sequence to actually open the file. Chances are that the user will extend greater effort in this endeavor than in the previous example, and thus his distance to the goal is greater. In both cases a distance exists, but one will certainly be shorter than the other. In other words, there necessarily

exists some distance between the establishment of a goal on the part of the user and his executing the action sequence required to meet that goal within the system. Minimizing this distance, which is referred to as the *Gulf of Execution*, is a key step towards establishing an interface that is easy to use.

A second aspect in determining the ease of an interface is establishing the amount of cognitive effort it takes for a user to perceive, interpret, and evaluate the computer's output that results from his actions. Again, there necessarily exists some distance between the generation of output on the part of the system and the mental steps required of a user to make sense of that output. For example, some windows based operating systems use different icons to distinguish between a file and a folder (a folder can "contain" additional files and folders). Upon "opening" a folder, a user can immediately determine the contents by looking at the icons. This is in contrast to many command-line, text based interfaces in which the names of files and the names of folders are indistinguishable (or at least harder to distinguish) than with the use of icons. In these systems, the text output given upon a user's request requires additional mental processing for the user to make determinations, and the amount of processing needed determines the distance to evaluation. This distance, which is referred to as the *Gulf of Evaluation*, should also be minimized to create an effective and easy to use UI.

Figure 3.1 illustrates the seven stages of user activities involved in the performance of a task for any human-computer system, as outlined by Hutchins et. al.

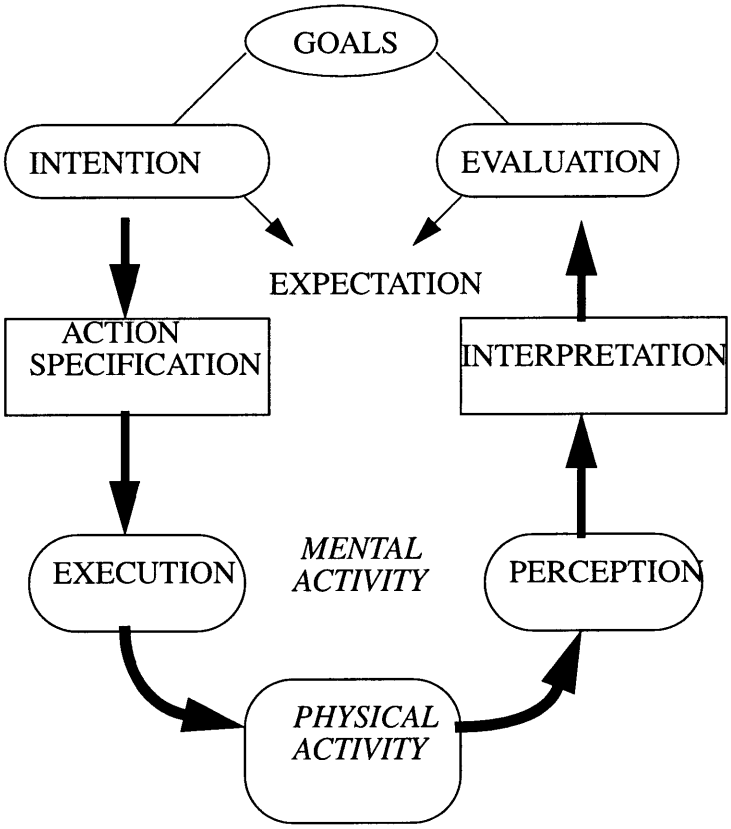

Figure 3.1: The seven stages of user activities involved in the performance of a task.

The Gulf of Execution and the Gulf of Evaluation can each be bridged from two directions. In one direction, the system can bridge the gaps through the interface by providing simple action sequences for the user to reach his goals and by providing output which is easy for the user to interpret. From the other direction, the user can bridge the gaps by forming mental constructs of the action sequences through practice. He can also form mental structures to deal with the output by learning to interpret the form in which it is given. Every form of human-computer interaction will undoubtedly involve a combination of both system based and user based bridging of these gaps. However, a simple and easy to use UI takes the burden off of the user as much as possible. Tasks which are practiced repeatedly can often become automated and may eventually require little effort; however, an interface which relies on the user in this manner and presents complex action

sequences will not be attractive to novice users, and will certainly not adhere to Nelson's *Ten Minute Rule*. Similarly, it is possible for users to *learn* the meaning of output which was once difficult to interpret and thus bridge the Gulf of Evaluation, but this too is undesirable if the goal is to build a UI that is easily accepted and widely used.

3.3 Direct Manipulation

> *The promise of Direct Manipulation is that instead of an abstract computational medium, all the 'programming' is done graphically, in a form that matches the way one thinks about the problem.*
>
> Hutchins, Hollans, and Norman, 1986 [7].

3.3.1 Background

Direct Manipulation is a type of interaction that takes place between a human and a computer system, facilitated by the form in which that system presents its data to the human, and the manner in which it allows a user to manipulate that data. One author describes Direct Manipulation as a process "in which the user manipulates, through a language of button pushes and movements of a pointing device such as a mouse, a graphic representation of the underlying data" [3]. The Macintosh Human Interface Guidelines (1992) state that "according to the principle of direct manipulation, an object on the [computer] screen remains visible while a user performs physical actions on the object ... [and] the impact of those operations on the object is immediately visible."

The term Direct Manipulation was coined by Ben Shneiderman (1983), who researched the virtues of interfaces that people enjoyed using versus the vices of those that users found frustrating and confusing [15]. Compiling the data from individuals who were enthusiastic users of their interactive systems, Shneiderman's reports were filled with positive feelings regarding:

> *mastery of the system*
> *competence in the performance of their task*

ease in learning the system originally and in assimilating advanced features

confidence in their capacity to retain mastery over time

enjoyment in using the system

eagerness to show it off to novices

desire to explore more powerful aspects of the system

Shneiderman developed a model of the features that produced such positive results and found the following central ideas recurring: visibility of the object of interest; rapid, reversible, incremental actions; and replacement of complex command language syntax by direct manipulation of the object of interest - hence the term "direct manipulation."

A good way to illustrate how Direct Manipulation actually works is to examine its use in many modern windows based operating systems. For example, the Macintosh operating system allows a user to move a computer file from one folder to another by using the mouse to click on an object that represents the file (a file icon), and drag this object to another object that represents the destination folder. To reverse this action, a user simply has to click the file icon again and drag it back to its original folder. The results of his actions are observed immediately in that the state of the system is instantly displayed with graphical objects that convey meaning through analogy and metaphor (Metaphors, in the UI sense, are discussed later in subsection 3.4.1). To accomplish these same tasks in a traditional command line interface, a user would have to manually enter text based commands which are usually operating system specific. Displaying the results of those actions would also require entering the proper text based commands to instruct the system to display its state.

3.3.2 Metaphors for Human-Computer Interaction

Hutchins et. al. discuss two major metaphors that exist for the nature of human-computer interaction: a conversation metaphor and a model world metaphor. The text based

interface discussed above is an example of the conversation metaphor. The interface is a syntactically structured language medium in which the user and the system have a conversation about the system state, where the system state is always hidden from the user. This interface is an intermediary which acts upon the system state and displays the results of those actions only when instructed to do so through proper use of syntax. The benefit is that the language provides an abstraction to the objects of interest (such as files and folders); however, it also acts as a barrier in that determinations about those objects can only be made through this specific language. In contrast, the Direct Manipulation style windows based operating system discussed above is an example of the model world metaphor. Here, the interface itself represents the world of the system's state. File icons and folder icons represent actual computer files and folders which reside within the system. User actions affect this world by manipulating objects on the screen, where the results of those actions become immediately apparent. In moving a file from one folder to another, the user can confirm visually that the appropriate file icon has changed position and be assured that the file now rests where he intended, thus removing the need for further inquiry to establish the results of his actions. Hence, the intermediary to the system state becomes transparent because users can act as if the interface *is* the world of the system state.

3.3.3 Evaluating Metaphors for Human-Computer Interaction

Referring back to section 3.2, the notions of the Gulf of Execution and the Gulf of Evaluation can be used to assess the model world metaphor, which is inherent to Direct Manipulation style interfaces, and the conversation metaphor, which describes the command line interfaces discussed above. The conversation metaphor imposes a structure on the user that is enforced by strict language syntax. The user cannot merely focus on his

task (i.e. moving a file), he must also process his knowledge of the computing domain (i.e. the command structure and syntax). This mental process increases the Gulf of Execution.

Additional syntactic inquiries are also required to determine how the user's actions have affected the system. Instead of being shown the resulting state immediately upon completing an action, the user must determine which system specific commands will yield information regarding the updated state and then type those commands to access this information. Furthermore, the resulting state will usually be displayed in text, forcing the user to make mental calculations to interpret what that text indicates. Thus, the Gulf of Evaluation is also increased in the conversation metaphor.

Direct Manipulation, on the other hand, relieves the user of specific knowledge of the computing domain and allows him to focus on the task at hand. For example, the action of moving a file is carried out metaphorically when the user moves the file icon on the screen. The results of the user's actions are displayed in such a way that little cognitive effort is needed to establish that the action was executed properly. By definition, an object remains on the screen while a user performs actions on it, and the impact of those operations is immediately visible. In these ways, both the Gulf of Execution and the Gulf of Evaluation are bridged by a Direct Manipulation interface, and much of the burden is removed from the user.

3.3.4 Advantages of Direct Manipulation

In his treatment of Direct Manipulation (DM), Shneiderman offers the following positive attributes which he found were consistent in most DM interfaces:

> *Novices can learn basic functionality quickly, usually through a demonstration by a more experienced user.*
>
> *Experts can work extremely rapidly to carry out a wide range of tasks.*

Error messages are rarely needed.

Users can immediately see if their actions are furthering their goals, and if not, they can simply change the direction of their activity.

Users experience less anxiety because the system is comprehensible and because actions are so easily reversible.

Users gain confidence and mastery because they initiate an action, feel in control, and can predict system responses. [15]

To support these claims, Shneiderman primarily examined various text editors that used DM, though he also used examples from Video Games, which are probably the most successful and popular applications of DM concepts. Anyone who has ever played a video game can readily understand the benefits of DM interfaces. In mid-1981 the author of *Video Invaders*, Steve Bloom, estimated that more than four billion quarters had been dropped into Space Invaders games around the world - that's roughly "one game per earthling." This popularity has certainly increased since the early 1980's with the advent of home entertainment systems and video games for home computers. One reason video game popularity has spread so fast is that novices can learn to play these games simply by watching someone else play. Shneiderman estimates that a beginner needs only 30 seconds to learn by watching someone else play, depending on the game. The ease of playing a video game comes from the fact that a user knows immediately whether he is reaching his goal, and can reverse his actions or try new options with the simple push of a joystick or button. For example, if he moves his spaceship too far to the left, he can simply push the joystick to the right to reverse the operation, or push forward or backward or at an angle to try something new. In a conversation metaphor type interface, this rapid generation of multiple alternatives is lacking.

Another one of the most popular advantages of a DM environment is that error messages are rarely needed since no syntax is required in the interface. A text editor in this

environment, for instance, allows the user to position the cursor by directly clicking (with a mouse) on the location where the cursor should be placed, or by guiding the cursor to that location with the arrow keys on the keyboard. However, moving the cursor of a text editor in the conversation metaphor would require converting a memorized physical action into correct syntactic form (such as UP6). This may be difficult to learn, hard to recall, and a source of frustrating errors. To take an example from basic programming, the user of a syntactic language interface could mistakenly misspell the name of an object and the system would produce an error message indicating that this object does not exist. But in a DM environment, the only objects that appear on the screen are those that actually do exist. Hence, an entire class of error messages is eliminated because the user can never point to a nonexisting object.

The discussion above illustrates how DM allows a user to simply and efficiently execute operations in the task domain without getting bogged down by details of the computing domain. Knowledge of the computer system or of computing becomes incidental, not required. Moving icons over other icons is effectively the same as writing programs or calling subroutines, with the advantage of being able to interact directly with the data and their semantic relationships, not the syntax for how those relationships must be expressed. In providing an interface that matches the way a user thinks about his tasks and goals, DM significantly reduces the amount of cognitive effort the user must expend.

3.3.5 Building Direct Manipulation Interfaces

Since Direct Manipulation is a fundamental aspect of the implementation described in this thesis, it is appropriate to briefly discuss the work that has been done to further the development of DM interfaces. Ideas hinting at Direct Manipulation programming and graphical interfaces have existed for over 20 years [Hutchins et. al., p. 91]. Hardware lim-

itations are the main reason these ideas did not find fruition earlier. But in recent times, with the computational power needed for graphical interfaces becoming cheap enough for wide availability, Direct Manipulation has become so popular that experts are researching the best techniques for building DM interfaces. In fact, many designers have started using Direct Manipulation interfaces for the very process of building them, as opposed to using conventional programming. This phenomenon alone suggests that Direct Manipulation is an extremely powerful style of human-computer interaction.

One such development tool is Peridot, which is a Direct Manipulation drawing package that guesses the relationships of new interface objects to existing ones using techniques from Artificial Intelligence [12]. Studies showed that nonprogrammers were able to create basic interface elements after about 1.5 hours of guided use with Peridot, while programmers were able to create more complicated interfaces in less than 10 percent of the time than they could using conventional programming. Other tools, such as Garnet, make it easy to create highly interactive, Direct Manipulation style interfaces by emphasizing easy specification of object behavior. This is done through a technique called Programming By Demonstration, which significantly reduces the need for conventional programming [10].

The question as to whether interfaces can be designed more effectively using conventional programming versus techniques like programming by example (as in the case of Garnet) or Direct Manipulation combined with Artificial Intelligence (as in the case of Peridot) remains an open one. Panel discussions regarding this issue have taken place with members contributing from both academia and industry [19]. What is apparent, however, is that Direct Manipulation style interfaces have gained significant popularity over the last two decades. Because they are among the most difficult to create, these interfaces

have demanded added attention from developers and researches who seek to change the way basic human-computer interaction takes place.

3.3.6 Summary

This section began by providing working definitions for Direct Manipulation, a term coined by Ben Shneiderman in 1983. Shneiderman's research indicated that DM style interfaces offered significant ease of use over traditional command line based interfaces. Concepts from Hutchins et. al. were introduced to validate the use of DM over syntactic languages as a means for manipulating data in a computer system. The benefits of Direct Manipulation were presented with examples from modern day windows based operating systems. Finally, a brief discussion was given of the work being done to further DM interface development.

3.4 Attributes of a Friendly User Interface

The human interface design principles outlined in this section have been taken primarily from the Macintosh Human Interface Guidelines. Though they are intended for use in designing products on Macintosh computers, these principles serve as a good set of guidelines for developing products on any platform since they are based on research done to determine "how people operate in the world" (p. 4). The subsection below (3.4.1) on Metaphors also makes references to a paper written by Erickson (1995) who is with the Advanced Technology Group at Apple Computer, Inc [8].

3.4.1 Metaphors

Erickson describes metaphor as an invisible yet ubiquitous part of our daily language and thought. Application designers can make use of this notion to build interfaces that provide functionality in a manner that is immediately familiar to a user. Concepts and fea-

tures of an application can be easily conveyed with an appropriate metaphor that takes advantage of people's knowledge about the world. For example, one widely used metaphor exploits people's familiarity with file folders. Computer documents are called files and are stored in objects called folders. In graphical interfaces, the computer folder is represented by an object that actually looks like a file folder. This use of metaphor immediately conveys functional information about how computer files can be stored, retrieved, moved, and discarded.

Erickson warns, however, that care must be taken when choosing a metaphor so that an inaccurate model does not mislead users by their understanding of the real world object. Voice mailboxes, for example, use the metaphor of a message being left in a physical box for retrieval by the owner at the owner's convenience. This model can fall short because in reality a message actually resides in a machine that could be miles away from the user. If the voice mail system is under heavy use, it can take quite a bit of time before the owner can retrieve his messages. Thus, a person may leave a message expecting that upon the recipient's return to his work desk he will immediately be able to hear the message and act accordingly, when in fact, a person may sit at his desk for 30 minutes before knowing the message was ever left. Erickson suggests that a better metaphor in this case would be an answering service metaphor in which messages are forwarded to the owner, and delays are understood beforehand since it is clear that messages do not actually sit in a box at the owner's desk.

Erickson goes on to suggest that in building good interface metaphors, designers should identify what functionality is most likely to be problematic for users. Care should be taken to find appropriate real-world events, objects, or institutions that embody some of the characteristics of the functionality that needs to be conveyed.

3.4.2 See-and-Point

The Macintosh Guidelines assert that users should be able to perform actions by interacting directly with the computer screen using a pointing device, typically a mouse. Two paradigms are exercised with See-and-Point interaction, both of which assume that users can always see the alternatives available to them on the screen and that they can always point to one of those alternatives.

In the first paradigm, a user selects an object from the screen and then selects the actions to be performed on the object. For instance, the Macintosh allows users to select a file icon and then select the "open" command from a menu. In the second paradigm, the user can drag an object onto some other object which performs an action associated with the first object. For example, the Macintosh desktop allows a user to drag a file icon to the trash object and thus discard the file.

3.4.3 Consistency

Consistency allows a user to transfer knowledge from one application or domain to other applications or domains. Doing so shortens the learning curve whenever a user switches between applications or between contexts in a given application. Consistent elements and operations are used to maintain standardized functionality where applicable. For example, different windows can contain different objects, but the same operations (or action sequences) are used on those objects to accomplish similar tasks (e.g., a file and a folder are both discardable by dragging the appropriate icon to the trash object).

3.4.4 WYSIWYG (What You See Is What You Get)

WYSIWYG environments make all features visually available to the user. Options are not hidden via abstract commands or hard to browse menus. If content must be hidden, little work should be required to access desired layers of that content. For example, rather than always displaying every file that resides recursively within a folder, it is often neces-

sary to use the abstraction of subfolders. Since opening a subfolder to display its contents requires very little additional effort, the user can easily view the content that is available to him.

3.4.5 User Control

Another principle important in designing interfaces is that the computer should not assume control by offering only those alternatives which are judged "good" for the user. Instead, the user should have freedom to initiate and control his actions and the interface should provide sufficient feedback to indicate the consequences of those actions. For example, if an open copy of a document resides somewhere on the Macintosh desktop, the user still has the freedom to initiate the action for deleting that document. Upon doing so, however, the system alerts the user with a dialog box that this file cannot be deleted until it has been properly closed. The system could have simply disallowed the user from initiating this action by making the file icon "undraggable," but that would have taken control away from the user. Maintaining User Control in an interface gives a person the ability to learn functionality through experimentation of various actions.

3.4.6 Feedback

A well designed system should use Feedback to keep users informed about what is happening in response to their actions. Such Feedback should be given as immediately as possible. Whenever a user initiates an action, the system should give some indication, visual or auditory, that the user's inputs have been received and are being processed. Lack of timely Feedback can leave a user wondering if he carried out his actions properly or whether those actions were ever interpreted by the system at all.

3.4.7 Forgiveness

A system which incorporates Forgiveness generally makes actions reversible, and makes good use of feedback to warn the user when certain irreversible actions can result in data loss. Forgiveness is closely relates to User Control in that it encourages users to

explore an application. Proper use of Forgiveness creates safety nets such that people feel comfortable learning and using a product.

3.5 Conclusion

This chapter discussed many of the issues which surround the design of an effective and efficient user interface. The argument was supported that a user interface should seek to minimize the amount of cognitive resources required of a user to accomplish his tasks. The Gulf of Evaluation and the Gulf of Execution were presented as two concepts through which one can assess the effectivess of a UI. Following that was a discussion on the two primary metaphors for human-computer interaction: the conversation metaphor and the model world metaphor. Direct Manipulation was presented as an interface style that subscribes to the model world metaphor, and an effective means of minimizing the user's cognitive efforts. Support for Direct Manipulation interfaces was given through a discussion of its popularity and the fact that advanced research is being conducted on the best ways to design and implement these types of interfaces. Finally, a description is given of attributes that should be present in all friendly user interfaces, as presented in the Macintosh Human Interface Guidelines. The concepts discussed in this chapter were integral in the design and construction of the GA Clock Interface, as will be seen in Chapter 4.

Chapter 4

The GA Clock Interface

The human-computer interface tool presented here allows users to perform efficient entry of data about events in time that are associated with temporal uncertainty. The technique allows for a single "drag-and-drop" operation to specify the type of event being recorded, the time at which the event occurred, and the user's confidence bounds on that time. Careful consideration was made in its design to create an effective, simple, and efficient means of human-computer interaction. Section 4.1 gives an overview of the GA Clock's functionality and the technology used to implement it. Section 4.2 discusses the specific issues involved with representing point and interval events with uncertainty. Section 4.3 describes how this tool incorporates the UI principles discussed in Chapter 3 to create an easy to use interface. Section 4.5 discusses some of the areas in which this interface could use improvements, and makes suggestions to that end. Finally, Section 4.6 describes the specific technology that was used in building the GA Clock Interface.

4.1 Overview

The GA Clock User Interface is implemented as a Java applet using Netscape's Internet Foundation Classes (IFC) as its primary code base. The IFC makes it easy to implement special purpose windows as well as drag and drop functionality. Three main windows comprise the overall interface: (i) the Source Window, in which icons represent various activities, (ii) the Clock Window, in which all temporal information associated with a given icon is displayed on the face of an actual clock, and (iii) a Trash Window, in which icons can be discarded (Figure 4.1).

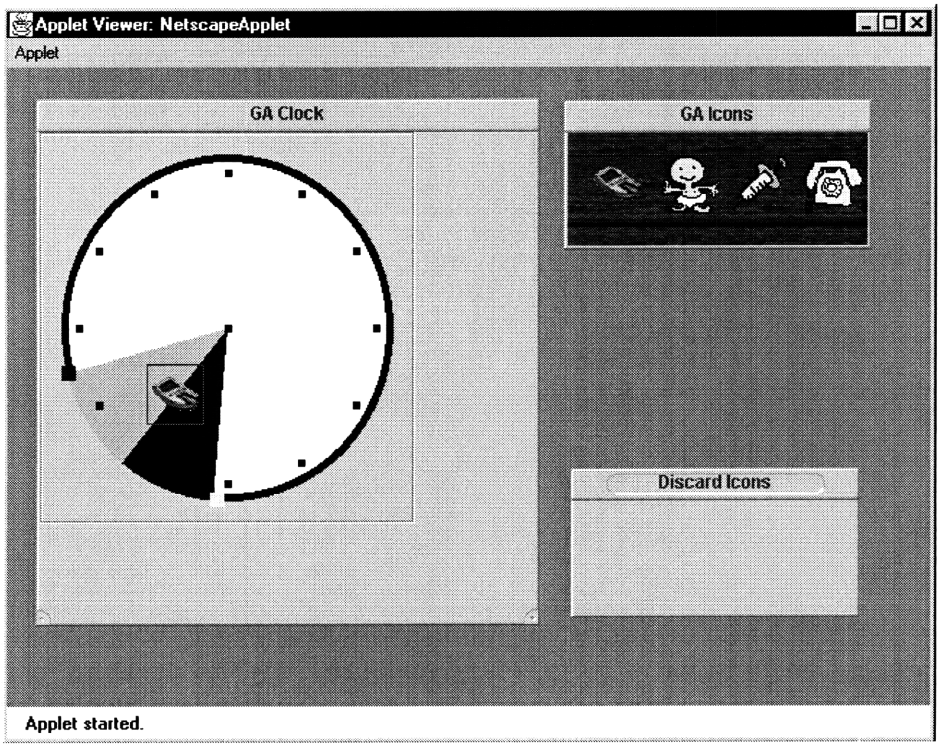

Figure 4.1: The GA Clock Interface is comprised of three windows: The Source Window (GA Icons), the Clock Window (GA Clock), and the Trash Window (Discard Icons).

The single compound action required to record the data can be broken down into the following distinct components: first, the mouse button is depressed over an icon of the item to be recorded and is dragged over a clock face. If dropped at the rim of the clock face, the event is recorded as occurring at exactly the corresponding time. If the icon is dragged into the face of the clock, an arc shows an interval of uncertainty that corresponds approximately to the arc subtended by the icon; thus, as the icon is dragged toward the center of the clock face, the uncertainty increases. The user always sees a graphical depiction of both the central time of the event and the arc of uncertainty, and can adjust both by the moving the icon. The action is completed when the icon is "dropped" (by releasing the mouse button). The icon is minimized in size at this point to indicate completion of an action. Subsequently re-selecting the icon allows it to be moved to indicate a revised time range and shows adjustment handles that permit moving the ends of the range of uncer-

tainty independently of the central time of the event; these adjustment handles allow the user to specify uncertainty bounds which are asymmetric around the central time.

4.2 Representing Events and their Temporal Uncertainty

This section describes how to represent the times of occurrence for point and interval events, along with their uncertainty. There is an inherent challenge in differentiating between point and interval events given the current implementation. This shortcoming of the interface is elaborated in subsection 4.2.3.

4.2.1 Point Events

Representing the time of occurrence for a point event that has no uncertainty is a simple and straightforward procedure. One motion is all that is needed to drag the appropriate event icon from the Source Window and drop it at the correct location in the Clock Window. After selecting the icon by depressing the mouse button, the user drags it towards the clock face and adjusts it radially (along the radius of the clock) and laterally (around the center of the clock) until he is satisfied that the position corresponds to the time of occurrence. If the icon is kept close to the rim of the clock, the arc subtended will be drawn as a straight line from the center of the clock to the rim, with its lateral position corresponding to the position of the icon. Adjusting the icon laterally allows the user to adjust where the line intersects with the rim, and thus depict the time of occurrence. When he is satisfied that the line points to the exact time of occurrence for the event, he should drop the icon at that location on the rim as shown in Figure 4.2.

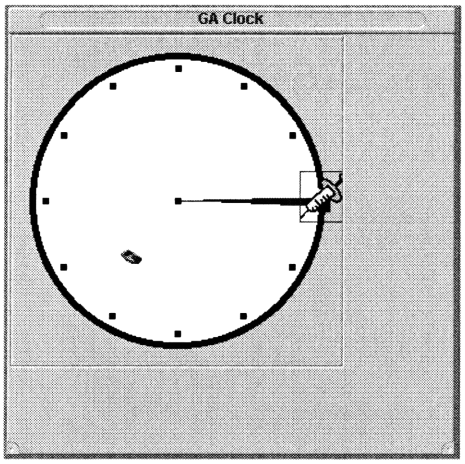

Figure 4.2: Representation of a Point Event with no uncertainty. The figure indicates that an injection was administered at exactly 3:00.

4.2.2 Point Events with Uncertainty

Representing a point event that has symmetric uncertainty is as simple as the procedure described above. However, it should be noted that for these types of events, the GA Clock does not distinguish between a point event with uniform uncertainty and a point event with non-uniform uncertainty (as described in subsection 2.3.3 and 2.3.4). The interface currently gives a means for viewing a central time within an uncertainty, but the mathematical interpretation of this probabilistic distribution is left to users of the system.

For representing point events with symmetric uncertainty, one motion will again suffice. Dragging the icon closer to the center of the clock increases the range of uncertainty associated with the event. Adjusting the icon laterally changes the center time within that interval. When the edges of the arc properly hit the boundaries of the interval of uncertainty, the icon can be dropped as shown in Figure 4.3. The center time can be determined by viewing the line which extends from center of the clock, through the icon, and out to the rim of the clock. This line appears because the two halves of the total arc drawn in different colors.

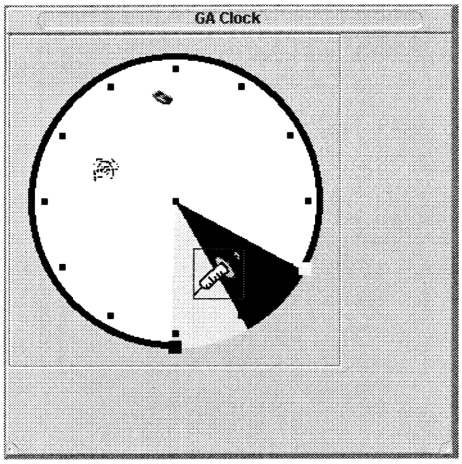

Figure 4.3: Representation of a Point Event with symmetric uncertainty. The figure indicates that an injection was administered sometime between 4:00 and 6:00, with a central time of 5:00.

Representing asymmetric uncertainty for a point event requires a small addition to the procedure described above for representing symmetric uncertainty. The user drags the icon over the clock face and establishes a "time of most likely occurrence" by adjusting the icon laterally. Radial motion will again increase or decrease the range of the uncertainty interval. Upon dropping the icon, the user can click the mouse button over one of the Uncertainty Handles, which appear at both edges of the arc of uncertainty, and drag this handle laterally to either increase or decrease the uncertainty on that side until he is satisfied with the interval boundaries. This time the line the drawn from the center of the clock and through the icon to the outer rim represents the "time of most likely occurrence." This line remains fixed after the initial drop of the icon. Figure 4.4 shows the final appearance of a point event with asymmetric uncertainty.

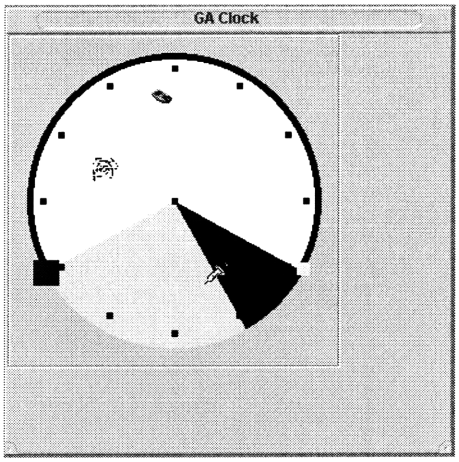

Figure 4.4: Representation of a Point Event with asymmetric uncertainty. The figure indicates that an injection was administered between 4:00 and 8:00, with 5:00 as the "time of most likely occurrence."

4.2.3 Interval Events and Uncertainty

Since the GA Clock Interface was designed specifically for representing point events and their uncertainty, the interface does not currently allow for the sufficient representation of interval events and their uncertainty. However, interval events *can* be represented if a few significant shortcoming are kept in mind. This subsection describes the procedure for representing interval events and the reasons why the current implementation is not complete in its treatment of these events. Section 4.4 discusses improvements that can be made to the existing interface to properly address the issue of representing interval events along with their uncertainty.

The procedure for representing interval events is inherently the same as for representing point events. This is also the reason that the current implementation is lacking - a point event and an interval event are essentially indistinguishable from one another in the GA Clock. Consider the example shown in Figure 4.5.

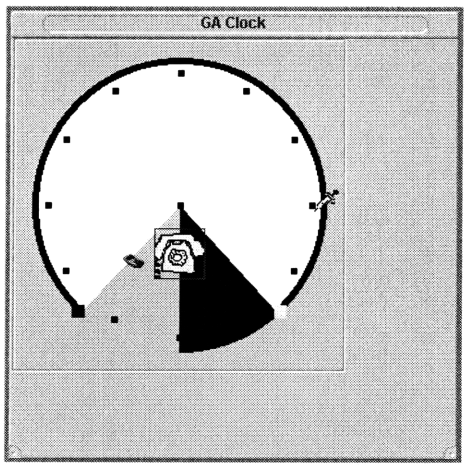

Figure 4.5: Representation of an Interval Event with no uncertainty. The figure represents a phone conversation that took place from 4:30 to 7:30.

The interval event is specified by dragging the icon to the clock face and adjusting its position to reflect an interval that begins with the start time of the event and ends with the end time of the event. Figure 4.5 shows that the duration for the event extends from approximately 4:30 to almost 7:30. However, it is not immediately apparent by way of the interface whether the event is a point event or an interval event. The example in Figure 4.5 could be misinterpreted as a point event occurring at the central time of 6:00, with 4:30 and 7:30 as the uncertainty bounds. Hence, one can only express an interval event if it is understood that the dropped icon represents an event that has duration, and not a point event. Furthermore, if the event's duration is absolutely known, but the start time and end time of the event is not exactly known, it is not possible to represent uncertainties for the these boundary times in the interface. Adjusting the handles extends what appear to be the bounds of the interval's duration. This adjustment merely creates the appearance of asymmetry because the "time of most likely occurrence" (which has no real meaning for an interval event) is still seen via the line which intersects with the clock rim.

To represent interval events properly, the interface must deal with interval event icons differently than it deals with point event icons. The specifics of this improvement are discussed further in Section 4.4.

4.3 Principles of UI Design in the GA Clock Interface

This section relates the GA Clock Interface to the issues on UI design presented in Chapter 3. Subsection 4.3.1 describes how the GA Clock fits the description of a Direct Manipulation style interface. An overall description of the features in the GA Clock is given as support. Subsection 4.3.2 discusses how different features of the interface relate to the principles of UI design presented in the Macintosh Human Interface Guidelines (discussed in section 3.4).

4.3.1 Direct Manipulation in the GA Clock

Shneiderman states that Direct Manipulation is a characteristic of interfaces that have the following properties:

1. *Continuous representation of the objects of interest.*
2. *Physical actions or labeled button presses instead of complex syntax.*
3. *Rapid incremental reversible operations whose impact on the object of interest is immediately visible.* (Shneiderman, 1982, p. 251) [16].

The primary objects of interest in the GA Clock are the Source Window, the Clock Window, the Trash Window, and the event icons that travel between them. These objects are never removed from the user's view of the screen. As an icon is dragged from one window to another, or even if it is being dragged aimlessly, it is always visible and appears under the constant control of the user. Whenever the user drops an icon into the Clock Window, it is minimized - not hidden or removed - to signify the completion of an action. The only time an icon warrants removal from the screen is when the user drops it into the Trash Window, signifying that this icon should be discarded.

The other 'object' which holds particular interest of the user is the arc subtended as an icon is dragged into the Clock Window. This arc shows the user how close he is to reaching his eventual goal in that it *is* the representation of temporal information. The arc is continuously drawn in real time to reflect the icon's changing position and disappears whenever the user removes the icon from the Clock Window. Removing the drawing of the arc at this time is appropriate in that an icon outside the Clock Window has no definable temporal information associated with it. The only other time this arc is removed is when the user selects a different icon to manipulate. At this stage, a new arc is drawn to represent the temporal information of the new icon. Whenever a previously positioned icon is re-selected, the arc of uncertainty is again drawn to reflect that icon's temporal information. Hence, all objects of interest are continuously displayed in the interface.

The GA Clock does not use any complex syntax in its current implementation. All communication and manipulation of data is done with physical actions via the mouse. To select an event icon, the user directs the mouse pointer to the Source Window and clicks the mouse button. To drag the icon, the user keeps the button depressed and moves the mouse. To release the icon, the user simply releases the button. The Uncertainty Handles are manipulated in exactly the same manner. Hence, the actions quickly become second nature and no command sequences or syntactic structures need to be memorized.

The redrawing of the uncertainty arc is done in real time, and hence the user can immediately see whether he is reaching his goal while manipulating an icon or moving one of the Uncertainty Handles. In the same way a video game allows for the rapid generation of new alternatives with a joystick, the GA Clock lets the user dictate with ease the directions in which he moves the object he is dragging. This means he can incrementally reverse the direction of movement, or try something completely new, just by pushing the mouse in a different direction. The display gives immediate feedback as to how his

actions are affecting the state of system. Hence, the GA Clock satisfies all three of Shneiderman's conditions for being classified as a Direct Manipulation interface.

4.3.2 Friendly UI Elements of the GA Clock

The ease in using the GA Clock comes from the simplicity of its display and the fact that feedback is given in real time. By making all relevant windows and icons completely visible to the user, the GA Clock has the quality of being a WYSIWYG interface. All objects are manipulated via straightforward See-and-Point operations, as has been described in previous sections. Dragging an icon object over another object such as the Clock Window or Trash Window performs an operation on one of those objects. For example, dragging an icon over the Trash Window discards the icon. This type of interaction falls under the second paradigm of See-and-Point operations (refer back to subsection 3.4.2).

The manipulation of icons from the Source Window is identical to manipulating icons within the Clock Window. This preserves the desirable trait of Consistency throughout the interface. The user actions required to move an Uncertainty Handle are also very similar to the way he drags icons, which reduces the amount of learning necessary to use the interface.

The user is given complete freedom to experiment with dragging icons and adjusting the uncertainty arc within the Clock Window. By appropriately combining User Control with Feedback and Forgiveness, the GA Clock encourages users to learn by doing. For example, when dragging an icon from the Source Window or when moving an icon from its original position in the Clock Window, the user has the freedom to take the icon anywhere on the computer screen. Eventhough dropping the icon in an area other than the Clock Window or Trash Window has no temporal meaning, the user still has the *freedom* to drag the icon wherever he wishes. This maintains User Control. But if the icon is ever

dropped in one of those undesignated areas, it immediately snaps back to the source from which it was originally dragged and the system appears as if the drag never took place. In essence, the user is given Feedback which states that dropping the icon in an undesignated area has no affect on the state of the system. This is also an appropriate use of Forgiveness since the user's action has not resulted in an un-reversible state of the system.

The use of real time Feedback is an essential part of manipulating the uncertainty arc within the Clock Window. When dragging the Uncertainty Handles, for instance, the user immediately knows if he is reaching his goal. However, if the user tries to drag one of the Handles all the way around the clock such that the arc of uncertainty would overlap itself, the interface prevents the drag from moving any further (see Figure 4.6). This is done to prevent the user from expressing an uncertainty that has no real temporal meaning, but it is done in a way that does not hinder the user from experimenting with different actions. This type of Feedback also satisfies conditions for Forgiveness in that disallowed actions are instantly communicated to the user

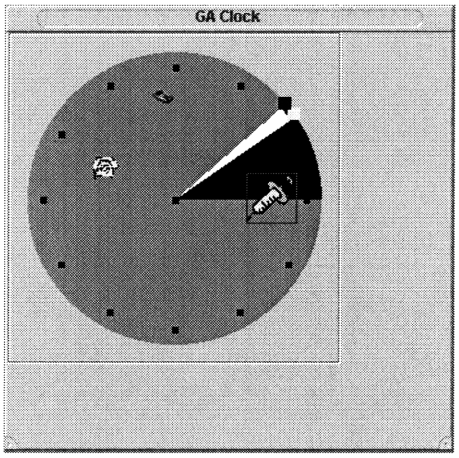

Figure 4.6: Overlapping of the uncertainty arcs is not allowed in the GA Clock Interface.

Additional uses of Feedback alert the user that the potential for action exists when he moves his mouse pointer to certain areas of the computer screen. Within in the Source Window, for instance, whenever the mouse travels over a source icon, the icon is high-

lighted with a white border to indicate the possibility of some action. Removing the mouse from this area causes the border to disappear. Within the Clock Window, whenever the mouse travels over a minimized icon, the icon is restored to its full size. This indicates to a user that *something* will happen if he clicks his mouse in this area - clicking that icon will make it the 'active' icon and its temporal information will be displayed. An Uncertainty Handle will also increase in size whenever the mouse travels over it, indicating that this is a valid object for mouse operations.

Finally, the use of Metaphor plays a key role in conveying the tasks that are handled in the GA Clock. There are two specific areas where Metaphor is used: the presence of appropriate event icons to represent occurrence of events, and the use of a clock face to represent temporal information. The current implementation of the GA Clock only uses a few icons, but one can imagine that in a live system there would be a wide range of icons available to the user. The use of a Syringe icon to indicate the taking of an injection, the use of a Glucometer icon to signify measuring ones blood sugar level, etc., are appropriate uses of Metaphor to convey meaning about the event. For any event that needs to be recorded via the GA Clock, an icon would be needed that pictorially conveys meaning about the event. An appropriate picture for the icon constitutes sufficient use of Metaphor in this case.

The clock face presents a more complicated use of Metaphor than do the icons. While the act of recording temporal information on a clock makes logical sense, this Metaphor has a functional problem associated with it. The GA Clock does not currently handle the problem of capturing temporal information about events that cross the A.M./P.M. border. If a user has recorded several events that took place throughout the day, he will encounter problems when entering events that took place around midnight. This is because the other icons which represent events from the P.M. time imply that crossing 12:00 means entering

the P.M. time range. But an event taking place around midnight needs to convey that crossing 12:00 now means going into A.M. time. The notion of AM time versus PM is not dealt with at all in the GA Clock.

This is a significant functional problem which cannot be addressed sufficiently in the current implementation. One possible solution would be to use a 24 hour clock instead of a 12 hour clock. But a similar problem would still exist when an event being recorded takes place around the midnight border between two days (the 24:00 to 00:00 border). The event icon must still indicate that it crossed the border of two separate days, while the clock itself only appears to cover the range of time in a given day. Thus, while the current Metaphor serves well as a proof of concept for showing that a single complex action can be used to enter information that is otherwise laborious to record, it falls short in a key area. Section 4.4 briefly discusses the use of a potentially more appropriate Metaphor.

4.4 Suggested Improvements on the Interface

Three major improvements to the interface are discussed in this section. The first is a method for better handling of interval events, which are not currently represented sufficiently. The second is a methodology for increasing the granularity of the temporal scale in the interface, and the reasons why this improvement is necessary. The third is a suggestion for a better metaphor than the clock face as an object on which to record temporal data.

The GA Clock currently treats all icons as if they represent point events with some central time or a time of most likely occurrence. This is because in its original design, the goal was only to represent point events. However, the representation of interval events along with point events seems necessary for a complete system. Therefore, interval event icons should be created which function differently from point event icons. Specifically,

when a user drags an interval event icon into the Clock Window, the arc subtended by the icon should appear in one solid color which represents the duration of the interval. For example, an icon representing the presence of a fever could be dragged into the clock, where a user could show that the fever lasted for four hours and the color of the interval is drawn in one solid color. Adjusting the icon's position should affect the duration interval the same way the uncertainty interval is adjusted for point events. By drawing the arc in a solid color, this event is differentiated from point events. Point events use two colors in the uncertainty arc primarily because the resulting line which separates the two colors is used to point to the "time of most likely occurrence." Since interval events have no similar notion of a central time, drawing the arc in a solid color does not cause the loss of valuable information, and this provides a visual means for differentiating interval events from point events. Furthermore, the Uncertainty Handles which are currently used to adjust asymmetric uncertainty, can be used indicate uncertainty around the interval of duration for an interval event. The color of these uncertainty arcs should be different from the color used to draw the interval of duration in order to separate known interval duration from the uncertainty intervals of the endpoints for the event.

The second major improvement involves increasing the granularity of the temporal scale in the interface. The user should be able to resize the Clock Window such that he can see minutes between the hour markers, and if desired, seconds within the minute markers. This type of variable resolution is necessary because many events being recorded need to be specified with the accuracy of minutes or even seconds. The current implementation only offers hour markers, which makes it difficult to specify that an event has an uncertainty interval between 9:55 and 10:05. Increasing the granularity would make this representation easier to capture.

Finally, a better metaphor than the clock face would probably be the use of a time line. A horizontal time line can function the same way as the clock in that proximity to the time line can determine the range of the interval. An icon can be dragged close to the time line to increase the interval range, and away from the time line to decrease the interval range. Such an implementation could be used to solve the current A.M./P.M. problem (as discussed in the previous section). A horizontal time line could extend for days, weeks, or even months in either direction, giving the user the ability to cross the borders between days without the confusion that exists in the clock face. A scroll bar could be used to quickly reach a desired portion of the time line. Uncertainty Handles could still be used to horizontally adjust the interval range in either direction of the time line to indicate asymmetry. Also, a separate resolution button could be added for the user to determine the desired granularity of the time line as he records his events.

4.5 Implementation Technology

The GA Clock Interface has been implemented in Java, which is an object-oriented language. Java was selected for its ability to run an application on different platforms without the need for code modification. The GA Clock is heavily based on the Internet Foundation Classes (IFC) from Netscape. The set of predefined classes found in IFC provide functionality for developers of windows based applications, making the IFC a natural choice for the features exhibited in the GA Clock.

Specifically, the IFC provides classes which readily implement many objects that can be used for graphical user interfaces, such as objects with drag-and-drop functionality. The IFC also makes it easy to build separate windows that are themselves draggable and resizable. Windows in the IFC can contain objects of the class View, which are equipped with event handling functions to capture button clicks and mouse movements that occur inside those window regions.

The remainder of this section describes the implementation of the two primary modules of the GA Clock Interface: the Source Window and the Clock Window. Specific components used to create these windows are described and the IFC's role in building the final application is elaborated.

4.5.1 The Source Window

The object in the Source Window which allows icons to be stored and viewed is called iconContainer. iconContainer is an instance of ContainerView, which is a class defined through the IFC. The ContainerView class allows one to define a region in which other View objects can be added, positioned, and displayed via the function addSubview(). The actual icon objects are added and displayed using addSubview().

The IFC class View allows the creation of rectangular objects which can track events such as mouse movements and button clicks that occur within their coordinate bounds. View objects can also be used to display bitmaps on the screen. Icon objects in the Source Window are instances of the class IconView, which inherits the functionality of the class View. Using the drawView() function provided for all View objects, IconView displays each icon's bitmap inside the icon's rectangular region. IconView also uses the functions mouseEntered() and drawView() to highlight its rectangular region whenever a mouse has entered the bounds of the icon.

Drag-and-drop functionality is implemented in IconView using functions from the IFC interface called DragSource, which allow IconView to create objects from the class DragSession. A DragSession object is created every time the mouse button is depressed while the mouse coordinates are contained within the region of the IconView object. DragSession objects follow mouse movement during the drag operation and are displayed using bitmaps provided in their constructor. IconView gives each DragSession object the same bitmap that it is currently using to display its icon. Thus, the icon being dragged looks the

same as the icon from which the drag was initiated. When the mouse button is released at the end of the drag operation, a DragSession object will check its coordinate position to establish whether it resides in a View that qualifies as a DragDestination. If it does, then the receiving View is alerted of the drag completion. Otherwise, the DragSession object terminates just after an animation that effectively snaps the bitmap display back to the View from which the drag began. This is why icons which are dropped in undesignated areas in the interface appear to immediately return back to the Source Window. The Clock Window implements the functions of the IFC interface DragDestination, allowing it to respond to drag operations that enter its coordinate region. This is explained further in the next section.

4.5.2 The Clock Window

The primary functions in the Clock Window are handled in an instance of the class ClockView. ClockView inherits the functionality of the IFC class ContainerView, thus allowing it to add Views as subviews and manipulate them. ClockView also implements the functions in the IFC interface called DragDestination. This allows ClockView to respond to DragSession objects which enter its coordinate region via the function dragEntered().

Implemented in ClockView is the function setClockParams(), which sets the parameters for drawing the subtended arc inside the clock face. This function is called from many different functions to handle the appearance of the clock face in various situations. dragEntered() calls setClockParams whenever a drag operation enters ClockView, and passes the coordinates of the DragSession object. Using these coordinates, setClockParams() calculates values for the width of the arc and its lateral position. It then passes these values to ClockView's function drawView(), which does the actual drawing.

As part of the interface for DragDestination, ClockView implements a function called dragDropped(), which is called whenever a drag operation is completed within ClockView's bounds. dragDropped() creates a new object from the class IconViewMovable. IconViewMovable is very similar to IconView but has a few added functions to support its role inside the Clock Window. To create an icon that looks identical to the object being dragged, dragDropped uses the bitmap from the DragSession object in the constructor for the IconViewMovable object. Hence, whenever an icon is dropped in the region of ClockView, a new icon object is created of type IconViewMovable which looks just like the icon that was dropped. This icon takes the position of the terminated DragSession object.

IconViewMovable implements DragSource and creates DragSession objects just like IconView does. However, if the completion of the drag operation occurs within ClockView, IconViewMovable uses the coordinates from the DragSession object to reposition itself. This gives the appearance that the icon actually moves to a new location when the drag is completed. Additionally, IconViewMovable uses state variables to store values that pertain to the position of the Uncertainty Handles. These values are updated whenever one of the Uncertainty Handles is manipulated. Storing these values is essential for maintaining the temporal information associated with each icon in the Clock Window.

Whenever the user switches between icons by clicking in the icon regions, IconViewMovable calls ClockView's function makeCurrentIcon() and passes the values for the position of the Uncertainty Handles for the icon being selected. These values are processed and the corresponding coordinates are sent to setClockParams() to initiate the proper drawing of the arc.

The Uncertainty Handles are also defined as Views, mainly for the purpose of capturing mouse movements. These objects are redrawn depending on whether or not the mouse

is in the object's coordinates. A given Uncertainty Handle can thus increase in size when there is a mouse over it, and minimize when the mouse leaves.

IconViewMovable also tracks mouse movement, maximizing the size of the icon when a mouse is in the icon region and minimizing when the mouse leaves.

4.5.3 Summary

The GA Clock Interface uses much of the predefined functionality given by Netscape's IFC. The other option considered was the Java Abstract Windowing Toolkit (AWT). Like the IFC, the AWT provides abstractions for windows and the handling of mouse events. However, the AWT did not provide native support for drag-and-drop functionality. Building this functionality from scratch was deemed unnecessary once the IFC was released.

The IFC's classes for Views, Windows, and DragSessions were integral in the GA Clock's development. No other set of library functions currently exist to rapidly create the functionality exhibited in the GA Clock.

4.6 Conclusion

The GA Clock Interface allows users to perform efficient entry of data about events in time that are associated with uncertainty. The Direct Manipulation style of this interface gives users the ability to represent complicated temporal information about point events with a single, complex action. Appropriate use of real time Feedback, User Control, and Forgiveness make for an interface that is easy to learn and simple to use. The interface can be improved by using a different Metaphor than the clock face for recording temporal data. Additional improvements would allow for more adequate representation of interval events.

This tool can be also be generalized to geometries other than the clock face or a time line to perform data entry for other types of information. For example, a range of oral temperatures could be entered by moving an icon of an oral thermometer to the vicinity of

a temperature scale; distance from the scale can indicate the size of the range of temperature.

For any item that is described by a choice among a small number of alternatives and a range of scalar values (like the temperature scale), the methods employed in the construction of the GA Clock can used to build an effective interface that allows data entry for those items.

Chapter 5

Conclusion

The GA Clock Interface was built in an effort to improve the means by which temporal data can be recorded in a computer. Because temporal data usually contains elements of uncertainty, representing that uncertainty is crucial for any system that makes analysis using temporal information. Bounded intervals have been suggested and used as a reasonable representation for the uncertainty that occurs in temporal data. Artificial Intelligence knowledge-based systems have made progress in temporal reasoning using intervals as a representation scheme.

Building a user interface that can account for all the variations on interval representations in not an easy task. Wide user acceptance of such an interface would require that all goals be simple to achieve and easy to comprehend. The GA Clock Interface is presented as a reasonable mechanism for representing point events and their uncertainty (symmetric or asymmetric). The benefit of the GA Clock over conventional interfaces (such as forms and menus) is that the user can engage in a single complex action to record information that is otherwise laborious to record. By providing a Direct Manipulation style of interaction, it is hoped that users will find using the interface easy and perhaps even enjoyable.

Appendix A

Java Code for GA Clock Interface

A.1 Clockifc.java

```
/*
    Notes - deciding how to implement a background view to display
some gif.  also noticed that
            redraws in any destination view are called from
iconViewMovable class, thus for the clock
            this may need to change since image of clock is not static
and redraws don't just occur
            within the clipping region defined by the moving icon
*/

import netscape.application.*;
import netscape.util.*;


public class Clockifc extends Application {
    public void init() {

        int                 x, width, height;

        InternalWindow      iconSupplies;          // The supplies
palette window
        ContainerView       iconContainer;

        IconView            icon;

        Image               iconImage;

        ClockWindow         clockWindow;
        ClockView           clockView;


        super.init();



        /* we want iconSupplies to be an InternalWindow because it
allows us to have a title bar,
           but we then insert iconContainer (a containerView) which
has a convenient method for
           the displaying of a background gif. */
```

```
        iconSupplies = new InternalWindow(mainRootView().width()/
2+100, 0, mainRootView().width()/3, 100);
        iconSupplies.setTitle("GA Icons");

        width = iconSupplies.contentView().width();
        height = iconSupplies.contentView().height();

        iconContainer = new ContainerView(-1, -1, width, height);
        iconContainer.setImageDisplayStyle(Image.TILED);
        iconContainer.setBorder(null);
        iconContainer.setImage(Bitmap.bitmapNamed("wood.gif"));

        clockWindow = new ClockWindow(120, 120, 350, 350);
        clockWindow.show();

        // Create, position, and add icons
        x = 10;

        iconImage = Bitmap.bitmapNamed("glucometer40.gif");
        icon = new IconView(x, 10, iconImage);
        iconContainer.addSubview(icon);

        iconImage = Bitmap.bitmapNamed("kid40.gif");
        icon = new IconView(x+50, 10, iconImage);
        iconContainer.addSubview(icon);

        iconImage = Bitmap.bitmapNamed("syringe40.gif");
        icon = new IconView(x+100, 10, iconImage);
        iconContainer.addSubview(icon);

        iconImage = Bitmap.bitmapNamed("Telephone2_40.gif");
        icon = new IconView(x+150, 10, iconImage);
        iconContainer.addSubview(icon);



        // Show the supplies
        iconSupplies.addSubview(iconContainer);
        iconSupplies.show();  // for any internalWindow use show()
rather than addSubview()

    }



    public static void main(String args[]) {
        Clockifc app;
        ExternalWindow mainWindow;
        Size size;

        app = new Clockifc();
        mainWindow = new ExternalWindow();
        size = mainWindow.windowSizeForContentSize(750, 650);
        mainWindow.sizeTo(size.width, size.height);
```

```
        mainWindow.show();
        app.setMainRootView(mainWindow.rootView());

        app.run();
    }
}
```

A.2 ClockWindow.java

```
import netscape.application.*;
import netscape.util.*;


public class ClockWindow extends InternalWindow {

    ClockView           clockView; //i'll refer to via a pointer


    public ClockWindow(int x, int y, int width, int height) {
        super(x, y, width, height);

        setResizable(true);
        setTitle("GA Clock");
        setAutoResizeSubviews(true);

        clockView = new ClockView(0, 0, 260, 260);
        clockView.setHorizResizeInstruction(View.WIDTH_CAN_CHANGE);
        clockView.setVertResizeInstruction(View.HEIGHT_CAN_CHANGE);

        contentView().addSubview(clockView);

        setHorizResizeInstruction(View.WIDTH_CAN_CHANGE);
        setVertResizeInstruction(View.HEIGHT_CAN_CHANGE);
    }

    public void didSizeBy (int deltaWidth, int deltaHeight) {
        super.didSizeBy(deltaWidth, deltaHeight);
       // contentView().setBounds(0, 0, width()-10, height()-10);
        //contentView().setBounds(10, 20, width()-20, height()-30);
        clockView.setBounds(0, 0, width()-20, height()-40);
        draw();
        System.out.println("called didSizeBy " + deltaWidth + "," +
deltaHeight);
    }

    public void setBounds (int x, int y, int width, int height) {

        super.setBounds (x, y, width, height);
        System.out.println("in setBounds " + x + "," + y);
        didSizeBy(x, y);
    }}
```

A.3 IconView.java

```
// IconView.java
// By Ned Etcode
// Copyright 1995, 1996 Netscape Communications Corp.  All rights
reserved.

import netscape.application.*;
import netscape.util.*;


public class IconView extends View implements DragSource {
    DragSession dragSession;          // A DragSession used for drag
'N drop
    boolean isHighlighted;
    int downX, downY;
    Image iconImage ;       //image used for dragging

    public IconView(int x, int y, Image bgImage) {
        super();
    //    setImage(bgImage);
        iconImage = bgImage;

        setBounds(x, y, 50, 50);
        setBuffered(true);
        draw();
    }


    public boolean isTransparent() {
        return true;
    }

    public View sourceView(DragSession session) {
        return this;
    }

    public void dragWasAccepted(DragSession session) {
        // Drag is complete
        dragSession = null;
    }

    public boolean dragWasRejected(DragSession session) {
        // Drag is complete, but rejected, so return true to animate
        // image back to initial location.
        dragSession = null;
        return true;
    }

    public void drawView(Graphics g) {

        iconImage.drawCentered(g, 0, 0, width(), height());
```

```
        if (isHighlighted()) {
            g.setColor(Color.lightGray);
            g.drawRect(0, 0, width(), height());
        }
    }

    public boolean mouseDown(MouseEvent event) {
            downX = event.x;
            downY = event.y;
        return true;
    }

    /** Start a drag session.
      */
    public void mouseDragged(MouseEvent event) {

        dragSession = new DragSession(this, iconImage,
                                      0, 0,
                                      downX, downY,
                                      "newIcon", iconImage);
    }

    public boolean wantsMouseTrackingEvents() {
        return true;
    }

    public void mouseEntered(MouseEvent e) {
        isHighlighted = true;
        draw();
    }

    public void mouseExited(MouseEvent e) {
        isHighlighted = false;
        draw();
    }

    public boolean isHighlighted() {
        return isHighlighted;
    }


}
```

A.4 IconViewMovable.java

```
// IconViewMovable.java

import netscape.application.*;
import netscape.util.*;


public class IconViewMovable extends View implements DragSource {
```

```
    DragSession     dragSession;          // A DragSession used for drag
'N drop
    boolean         isHighlighted, is_small;
    int             downX, downY;
    Image           iconImage ;       //image used for dragging

    Point           uncPoint1, uncPoint2;

    Rect            lastBounds;


    public IconViewMovable(Rect bounds, Image bgImage) {
        super(bounds);

        iconImage = bgImage;

        uncPoint1 = new Point();
        uncPoint2 = new Point();

        setBuffered(true);
        is_small = true;

        draw();
    }


    public boolean isTransparent() {
        return true;
    }

    public View sourceView(DragSession session) {
        return this;
    }

    public void dragWasAccepted(DragSession session) {

        Rect            oldPosition;
        ClockView       myClockView;

        // Drag is complete

        oldPosition = bounds();

        moveTo (session.destinationBounds().x,
                session.destinationBounds().y);

        myClockView = (ClockView)superview();
        myClockView.updateIcon(this);

        myClockView.draw(oldPosition);      // needed to vanish old
iconImage in clock's view

        dragSession = null;
    }
```

```
    public boolean dragWasRejected(DragSession session) {
        // Drag is complete, but rejected, so return true to animate
        // image back to initial location.
        dragSession = null;
        return true;
    }

    public void drawView(Graphics g) {

        if (is_small) {
            iconImage.drawScaled(g, width()/4, height()/4, width()/2,
height()/2);
        } else {
                iconImage.drawCentered(g, 0, 0, width(), height());
        }

        if (isHighlighted()) {
            g.setColor(Color.darkGray);
            g.drawRect(0, 0, width(), height());
        }
    }

    public boolean mouseDown(MouseEvent event) {
        ClockView   myClockView;

        myClockView = (ClockView)superview();
        myClockView.makeCurrentIcon(this);

        return true;
    }

    /** Start a drag session.
      */
    public void mouseDragged(MouseEvent event) {

        dragSession = new DragSession(this, iconImage,
                                      0, 0,
                                      downX, downY,
                                      "movingIcon", this);
    }

    public boolean wantsMouseTrackingEvents() {
        return true;
    }

    public void mouseEntered(MouseEvent e) {
        isHighlighted = true;
        is_small = false;
        draw();
    }

    public void mouseExited(MouseEvent e) {
        isHighlighted = false;
```

```
        is_small = true;
        draw();
    }

    public boolean isHighlighted() {
        return isHighlighted;
    }


}
```

A.5 ClockView.java

```
import netscape.application.*;
import netscape.util.*;
import java.lang.Math;


public class ClockView extends ContainerView implements
DragDestination {

    public static final double RADIANS_PER_MINUTE = Math.PI / 30;

    int                 i;
    Point               clkCenter, uncPoint1, uncPoint2;
    Rect                clkBounds;      // the clock's rectangular bounds
    int                 clkRadius;

    boolean             isUncDrag1, isUncDrag2;

    Rect                uncRect1, uncRect2,
                        uncRect1_sm, uncRect2_sm,
                        uncRect1_big, uncRect2_big;
    double              uncAng1_deg, uncAng2_deg;

    Color               color1, color2;

    double              theta_deg, theta_rad;
    double              fillArc1_deg, fillArc2_deg; // two arcs in draw

    double              ratio;          // aspect ratio
    int                 space;          // spacing value
    Point               temp;

    TextField           statusField;

    IconViewMovable     currentIcon;


    public ClockView(int x, int y, int width, int height) {
        super(x, y, width, height);

        setBackgroundColor(Color.lightGray);
```

```
    temp = new Point();
    clkCenter = new Point();
    clkBounds = new Rect(15, 15, width-30, height-30);
    clkCenter.x = clkBounds.midX();
    clkCenter.y = clkBounds.midY();
    clkRadius = clkBounds.width/2;

    ratio = clkBounds.width/clkBounds.height;
    space = (clkBounds.height/40) < 2 ? 2 : clkBounds.height/40;

    uncPoint1 = new Point();
    uncPoint2 = new Point();
    uncRect1 = new Rect();
    uncRect2 = new Rect();
    uncRect1_sm = new Rect();
    uncRect2_sm = new Rect();
    uncRect1_big = new Rect();
    uncRect2_big = new Rect();

    theta_deg = 0;
    fillArc1_deg = 0;
    fillArc2_deg = 0;
    uncAng1_deg = 0;
    uncAng2_deg = 0;

    /*movedFrom = new Point();
    movedFrom.x = 0;
    movedFrom.y = 0;*/

    currentIcon = null;

    // Use buffering for smoother animation
    setBuffered(true);

}



public void mouseMoved (MouseEvent event) {

    if (uncRect1_big.contains(event.x, event.y)) {
        uncRect1 = uncRect1_big;
        draw(uncRect1_big);
    } else {uncRect1 = uncRect1_sm;
            draw(uncRect1_big);
           }

    if (uncRect2_big.contains(event.x, event.y)) {
        uncRect2 = uncRect2_big;
        draw(uncRect2_big);
    } else {uncRect2 = uncRect2_sm;
            draw(uncRect2_big);
           }
```

```
    }



    public boolean mouseDown(MouseEvent event) {

        if (uncRect2_big.contains(event.x, event.y)) {
            isUncDrag2 = true;
            isUncDrag1 = false;
            return true;
        }   else {
            if (uncRect1_big.contains(event.x, event.y)) {
                isUncDrag1 = true;
                isUncDrag2 = false;
                return true;
            }   else {

                isUncDrag1 = false;
                isUncDrag2 = false;

                currentIcon = null;
                setClockParams(event.x - clkCenter.x, event.y -
clkCenter.y);
                draw();
                return true;
            }
        }
    }


    public void mouseDragged (MouseEvent event) {

        int             deltaX, deltaY;

        deltaX = event.x - clkCenter.x;
        deltaY = event.y - clkCenter.y;

        if (isUncDrag1) {
            aSymmetricParams(uncRect1_big, deltaX, deltaY);
            if (currentIcon != null) {updateIcon(currentIcon);}
            draw();
        }   else {
            if (isUncDrag2) {
                aSymmetricParams(uncRect2_big, deltaX, deltaY);
                if (currentIcon != null) {updateIcon(currentIcon);}
                draw();
            }   else {

                setClockParams(deltaX, deltaY);
                draw();
```

```
            }
        }
    }

    public void setDirection (Point movedTo) {
    }

    public void drawView(Graphics g) {

        drawViewSecond(g);
    }

    public void drawViewSecond(Graphics g) {

        g.setColor(Color.lightGray);
        g.fillRect(0, 0, this.width(), this.height());

        g.setColor(Color.black);
        g.fillOval(clkBounds);

        g.setColor(Color.white);
        g.fillOval(
                    clkBounds.x + space,
                    clkBounds.y + space,
                    clkBounds.width-space*2,
                    clkBounds.height-space*2);



        //drawClockFace(g);

        color1 = new Color((int)(fillArc1_deg/1.0));
        g.setColor(color1);
        g.fillArc(clkBounds, (int)theta_deg, (int)fillArc1_deg);

        color2 = new Color((float)(fillArc2_deg/1.0),
(float)(fillArc2_deg/1.0),
                           (float)(fillArc2_deg/1.0));
        g.setColor(color2);
        g.fillArc(clkBounds, (int)theta_deg, (int)fillArc2_deg);

        g.setColor(Color.black);
        for (int c = 1; c < 13; c++)
        {
            clockSurface(c * 5,
                         clkCenter.y - (int)(clkBounds.y + space*2.5) );
            g.fillRect(
                temp.x - space/2, temp.y - space/2,
                space, space);

        }

        g.fillRect(clkCenter.x-space/2, clkCenter.y-space/2,
            space, space);
```

```
        g.setColor(Color.yellow);
        g.fillRect(uncRect1);
        g.setColor(Color.blue);
        g.fillRect(uncRect2);



    }

    public void drawClockFace(Graphics g) {
        g.setColor(Color.black);
        g.fillOval(clkBounds);

        g.setColor(Color.white);
        g.fillOval(
                    clkBounds.x + space,
                    clkBounds.y + space,
                    clkBounds.width-space*2,
                    clkBounds.height-space*2);

        g.setColor(Color.black);
        for (int c = 1; c < 13; c++)
        {
            clockSurface(c * 5,
                         clkCenter.y - (int)(clkBounds.y + space*2.5) );
            g.fillRect(
                temp.x - space/2, temp.y - space/2,
                space, space);

        }

        g.fillRect(clkCenter.x-space/2, clkCenter.y-space/2,
            space, space);
    }




    /** We call showStatus() from several places to update the text
displayed
      * in the status field.
      */
    public void showStatus(String status) {
        statusField.setStringValue(status);
    }


    public boolean isTransparent() {
                return false;
    }

    public DragDestination acceptsDrag(DragSession session, int x, int
```

```
y) {
        // If an icon is being dragged to us, return the DragDestination
        // (the TankView itself.)


        if  ( ("movingIcon".equals(session.dataType())) |
              ("newIcon".equals(session.dataType())) )
          {
           return this;
            }

        else {
            return null;
        }
    }

    public boolean dragDropped(DragSession session) {

        //NOTE:  for some reason redrawing in this method did not clear
        //       the iconImage from moved position - instead, called
draw
        //       from dragAccepted method of iconViewMovable

        IconViewMovable       item;
        Image                 iconImage;

        if ("newIcon".equals(session.dataType()))
            {
                iconImage = (Image)session.data();
                item = new IconViewMovable(session.destinationBounds(),
                                           iconImage);
                updateIcon(item);
                currentIcon = item;
                addSubview(item);

                return true;
            }
            else
            {
                if ("movingIcon".equals(session.dataType()))
                  {
                    return true;
                  }
                  else { return false;}
            }

    }

    public void makeCurrentIcon (IconViewMovable icon) {

        int deltaX, deltaY, deltaX1, deltaY1, deltaX2, deltaY2;

        currentIcon = icon;
```

```
    deltaX = icon.bounds().midX() - clkCenter.x;
    deltaY = icon.bounds().midY() - clkCenter.y;
    theta_rad = calcThetaRad(deltaX, deltaY);
    theta_deg = theta_rad*180/Math.PI;

    deltaX1 = icon.uncPoint1.x - clkCenter.x;
    deltaY1 = icon.uncPoint1.y - clkCenter.y;
    uncAng1_deg = 180/Math.PI*(calcThetaRad(deltaX1, deltaY1));


    deltaX2 = icon.uncPoint2.x - clkCenter.x;
    deltaY2 = icon.uncPoint2.y - clkCenter.y;
    uncAng2_deg = 180/Math.PI*(calcThetaRad(deltaX2, deltaY2));

    aSymmetricParams(uncRect1_big, deltaX1, deltaY1);
    aSymmetricParams(uncRect2_big, deltaX2, deltaY2);


    draw();
}

public void updateIcon (IconViewMovable icon) {
    icon.uncPoint1.x = uncPoint1.x;
    icon.uncPoint1.y = uncPoint1.y;
    icon.uncPoint2.x = uncPoint2.x;
    icon.uncPoint2.y = uncPoint2.y;
}

public boolean dragEntered(DragSession session) {

    return true;
}

public void dragExited(DragSession session) {

    System.out.println("in dragExited, calling setClockParams");
    theta_deg = 0;
    fillArc1_deg = 0;
    fillArc2_deg = 0;
    uncAng1_deg = 0;
    uncAng2_deg = 0;

    uncRect1_sm = new Rect();
    uncRect2_sm = new Rect();


    draw();

}


public boolean dragMoved(DragSession session) {

    int             deltaX, deltaY;
```

```
        deltaX = session.destinationBounds().midX() - clkCenter.x;
        deltaY = session.destinationBounds().midY() - clkCenter.y;

        setClockParams(deltaX, deltaY);

        draw();

        return true;
    }


    public void setUncRect(double uncAngle_rad, int which_rect) {

        if (which_rect == 1) {

            uncPoint1.x =
(int)(clkRadius*Math.cos(uncAngle_rad))+clkCenter.x;
            uncPoint1.y = (int)(clkRadius*Math.sin(uncAngle_rad))*(-
1)+clkCenter.y;

            uncRect1_sm = new Rect(uncPoint1.x - space, uncPoint1.y -
space, 2*space, 2*space);
           uncRect1_big = new Rect(uncPoint1.x - 2*space, uncPoint1.y -
2*space, 4*space, 4*space);

            uncRect1 = uncRect1_sm;
            uncAng1_deg = uncAngle_rad*180/Math.PI;
        }
        else {
            if (which_rect == 2) {

                uncPoint2.x =
(int)(clkRadius*Math.cos(uncAngle_rad))+clkCenter.x;
                uncPoint2.y = (int)(clkRadius*Math.sin(uncAngle_rad))*(-
1)+clkCenter.y;

                uncRect2_sm = new Rect(uncPoint2.x - space,
uncPoint2.y - space, 2*space, 2*space);
                uncRect2_big = new Rect(uncPoint2.x - 2*space,
uncPoint2.y - 2*space, 4*space, 4*space);

                uncRect2 = uncRect2_sm;
                uncAng2_deg = uncAngle_rad*180/Math.PI;

            } else {System.out.println("no such rect");}
        }
    }

    public double calcThetaRad (int deltaX, int deltaY) {

        double          hypotenuse;
        double          radian_angle;
```

```
        hypotenuse = Math.sqrt( (deltaX*deltaX) + (deltaY*deltaY) );
        deltaY = -deltaY;  //get back to a normal coord system

        if (deltaX >= 0) { //if in quad I or IV
            radian_angle = Math.asin(deltaY/hypotenuse);}
            else {
                    if (deltaY<0) { //if in quad III
                        deltaY = -deltaY;
                        radian_angle = Math.asin(deltaY/hypotenuse) +
Math.PI;}
                    else { //in quad II
                        radian_angle = Math.PI - Math.asin(deltaY/
hypotenuse);}
            }
        if (radian_angle < 0) {radian_angle = radian_angle + 2*Math.PI;}
        return (radian_angle);
    }


   /* this procedure sets theta and arcAngle for a clock redraw */

   public final void setClockParams(int deltaX, int deltaY) {
        double      arcAngle_rad, arcAngle_deg;
        double      uncAngle1, uncAngle2; //used to calc uncRects
        double      hypotenuse;

        hypotenuse = Math.sqrt( (deltaX*deltaX) + (deltaY*deltaY) );

        theta_rad = calcThetaRad(deltaX, deltaY);

        theta_deg = theta_rad*180/Math.PI; // at this point i have
correct theta_deg

        fillArc1_deg = 180*(1 - Math.exp( (-1)*(width()/2-
hypotenuse)*.003 ));
        fillArc2_deg = -fillArc1_deg;

        arcAngle_rad = fillArc1_deg*Math.PI/180;

        uncAngle1 = theta_rad + arcAngle_rad;
            if (uncAngle1 < 0) {uncAngle1 = uncAngle1 + 2*Math.PI;}//
don't know whether
        uncAngle2 = theta_rad - arcAngle_rad;
            if (uncAngle2 < 0) {uncAngle2 = uncAngle2 + 2*Math.PI;}//I
need these if's.

        setUncRect(uncAngle1, 1);
        setUncRect(uncAngle2, 2);

   }

    public void aSymmetricParams(Rect uncRect,
                                 int deltaX,
```

```
                                int deltaY) {

        double      uncTheta_rad, uncTheta_deg;

        uncTheta_rad = calcThetaRad(deltaX, deltaY);
       uncTheta_deg = uncTheta_rad*180/Math.PI; // at this point i have
correct theta_deg

        if (uncRect.equals(uncRect1_big)) {
            if (uncTheta_deg > theta_deg) {
                if ( (uncAng2_deg > theta_deg)&&
                     (uncTheta_deg > (uncAng2_deg-5))){
                      uncTheta_deg = uncAng2_deg-5;
                      uncTheta_rad = uncTheta_deg*Math.PI/180; }

                fillArc1_deg = uncTheta_deg - theta_deg;}
                else if (theta_deg > uncTheta_deg) {
                    if (uncTheta_deg > (uncAng2_deg-5)) {
                        uncTheta_deg = uncAng2_deg-5;
                        uncTheta_rad = uncTheta_deg*Math.PI/180;
                        fillArc1_deg = 360 - theta_deg + uncTheta_deg;}

                    if (uncAng2_deg > theta_deg) {
                        uncTheta_deg = uncAng2_deg-5;
                        uncTheta_rad = uncTheta_deg*Math.PI/180;
                        fillArc1_deg = uncTheta_deg - theta_deg;}
                    else {
                        fillArc1_deg = 360 - theta_deg + uncTheta_deg;}

                        }
            setUncRect(uncTheta_rad, 1);


        }
            else if (uncRect.equals(uncRect2_big)) {
                    if (theta_deg > uncTheta_deg) {
                       if ((theta_deg > uncAng1_deg) &&
                           (uncTheta_deg < (uncAng1_deg+5))) {
                            uncTheta_deg = uncAng1_deg+5;
                            uncTheta_rad = uncTheta_deg*Math.PI/180;}

                       fillArc2_deg = uncTheta_deg - theta_deg;}

                    else
                     if (uncTheta_deg > theta_deg) {

                        if (uncAng1_deg < theta_deg) {
                            uncTheta_deg = uncAng1_deg+5;
                            uncTheta_rad = uncTheta_deg*Math.PI/180;
                            fillArc2_deg = uncTheta_deg - theta_deg;}
                        else {
                                if (uncTheta_deg < (uncAng1_deg+5)) {
                                    uncTheta_deg = uncAng1_deg+5;
                                    uncTheta_rad = uncTheta_deg*Math.PI/
```

```
180;}

                                fillArc2_deg = (-1)*(theta_deg + 360 -
uncTheta_deg);}}
                    setUncRect(uncTheta_rad, 2);
            }


                                }




   public final void clockSurface(int minutes, int length)
   {
        minutes = (75 - minutes) % 60;
        double x = Math.cos(minutes * RADIANS_PER_MINUTE) * length;
        double y = Math.sin(minutes * RADIANS_PER_MINUTE) * length;
        temp.x = clkCenter.x + (int)(x * ratio);
        temp.y = clkCenter.y - (int)y;
   }

}
```

A.6 NetscapeApplet.java

```
// NetscapeApplet.java
// By Ned Etcode
// Copyright 1995, 1996 Netscape Communications Corp.  All rights
reserved.

import netscape.application.*;
import netscape.util.*;

public class NetscapeApplet extends FoundationApplet {
    /** This method must be implemented by the applet developer because
      * there is no way in the standard Java API for system classes
(like
     * netscape_beta.application) to look up an applet's class by name.
The
      * static method Class.ForName() simply looks up one level in the
stack
      * and gets the ClassLoader associated with the method block of the
     * caller. When the netscape_beta.application classes are installed
as
      * system classes their ClassLoader is null, so when code in
      * netscape_beta.application calls Class.forName() it can only find
      * other system classes. What is needed is API which allows code to
      * find the ClassLoader for an applet by URL, and public API on
      * ClassLoader to ask it to load classes by name. Until those
      * enhancements can be made and distributed in all the java systems
      * in the world, applets will have to subclass FoundationApplet and
```

```
     * implement this one-line method:
     * <pre>
     *     public abstract Class classForName(String className)
     *         throws ClassNotFoundException {
     *         return Class.forName(className);
     *     }
     * </pre>
     */
    public Class classForName(String className)
        throws ClassNotFoundException {
        return Class.forName(className);
    }
}
```

References

[1] Allen, James F. "Maintaining Knowledge About Temporal Intervals," *Communications of the ACM*, vol. 26, no. 11, 1983.

[2] Apple Computer, Inc. *Macintosh Human Interface Guidelines.* Addison-Wesley, 1992.

[3] Baecker, Ronald M., Buxton, William A.S., *Readings in Human-Computer Interaction: A Multidisciplinary Approach*, Morgan-Kaufman, 1987.

[4] Bobrow, D.G., Mittal, S., Stefik, M.J., "Expert Systems: Perils and Promise," *Communications of the ACM*, 29(9), pp. 880-894, 1986.

[5] Cardelli, Luca., "Building User Interfaces by Direct Manipulation," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta, Canada, Oct. 1988, pp. 152-166.

[6] Cousins, Steve B., Kahn, Michael G., "The Visual Display of Temporal Information," *Artificial Intelligence in Medicine*, pp. 341-357, 1991.

[7] Hutchins, Edwin L., Hollan, James D., Norman, Donald A., "Direct Manipulation Interfaces," in: Norman and Draper, eds., *User Centered System Design - New Perspectives on Human-Computer Interaction*, pp. 87-124, 1986

[8] Erickson, Thomas D., "Working with Interface Metaphors," *Readings in HCI: Toward the Year 2000* (2nd Edition), pp. 147-151.

[9] Kohane, Isaac S., "Temporal Reasoning In Medical Expert Systems," Ph.D. Thesis, Massachusetts Institute of Technology, 1987.

[10] Myers, Brad A., Giuse, Dario A., Dannenberg, Roger B., Zanden, Brad V., Kosbie, David S., Pervin, Edward., Mickish, Andrew., Marchal, Philippe., "Garnet: Comprehensive Support for Graphical Highly Interactive User Interfaces," *IEEE Computer* 23(11), pp. 71-85, Nov. 1990.

[11] Myers, Brad A., Zanden, Brad V., "Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces," *Human Factors in Computing Systems CHI '90 Conference Proceedings Special Issue of the SIGCHI Bulletin*, pp. 27-34, April 1990.

[12] Myers, Brad A., "Using AI Techniques To Create User Interface By Example," *Intelligent User Interfaces* (Sullivan and Tyler), pp. 385-401, 1991.

[13] Nelson, T., *Ten Minute Rule*, p. DM58, 1974.

[14] Russ, Thomas A., "Reasoning With Time Dependent Data," Ph.D. Thesis, Massachusetts Institute of Technology, 1991.

[15] Shneiderman, Ben., "Direct Manipulation: A Step Beyond Programming Languages," *Computer*, 16(8), pp. 57-69, August 1983.

[16] Shneiderman, Ben., "The Future of Interactive Systems and the Emergence of Direct

Manipulation," *Behaviour and Information Technology*, pp. 237-256, 1982.

[17] Sutton, J.A., Sprague, R.H., Jr., "A Study of Display Generation and Management in Interactive Business Applications," *Technical Report RJ2393*, IBM Research Report, Nov. 1978.

[18] Szolovits, Peter., Doyle, Jon., Long, William., *Guardian Angel: Patient-Centered Health Information Systems* Laboratory for Computer Science, May 1994.

[19] Wiecha, Charles.,"Direct Manipulation Or Programming: How Should We Design Interfaces?" *Proceedings of the ACM Symposium on User Interface Software and Technology*, Williamsburg, VA, USA, pp. 124-126, Nov. 13-15, 1989.

[20] Wiederhold, G., Fries, J.F., Weyl, S., "Structured Organization of Clinical Databases," in: D.A. Meier and S.W. Miller, eds., *AFIPS Conference Proceedings 44*, pp. 479-485.

[21] Winston, Patrick H., *Artificial Intelligence*, 3rd Edition, Addison-Wesley, pp. 265-275, 1992.