Tool Support for Component-Based Programming

by

Catherine Amy Morton

Technical Report No. CS-94-02

1994

Department of Computer Science
York University
North York, Ontario
Canada M3J 1P3

Tool Support for Component-Based Programming

Catherine Amy Morton

A thesis submitted to the Faculty of Graduate Studies in partial fulfillment of the requirements for the degree of

Master of Science

Graduate Programme in Computer Science
York University
North York, Ontario

May 1994

Abstract

Component-based programming refers to a system in which applications are built by creating frameworks of software components which interact with each other. Component-based programming has the potential to overcome many of the problems currently associated with the development of interactive applications. The most important aspect of a component-based application is its component structure or architecture. These structures become large and complex as applications grow. Developers are not able to manage this complexity without appropriate tool support.

This thesis makes use of Clock, a component-based system designed for the creation of interactive applications. In Clock, programs are structured as a hierarchy of components. A detailed task analysis of the Clock programmer was conducted to identify the requirements for the development of a programming tool to support the task of creating and manipulating Clock architectures. The first version of the ClockWorks (the new Clock Programming Environment) has been implemented and tested with its users. It provides graphical representation and direct manipulation of Clock architecture structures.

Results of user testing and evaluations of *ClockWorks* illustrate that it has met seven of the ten requirements identified. It has been successful in assisting developers in the task of creating and managing Clock architecture structures. The ideas used in the creation of *ClockWorks* could be modified so that they could apply to other component-based programming systems.

Acknowledgments

I would not have been able to do any of this work without the help of my supervisor Nick Graham. Thank you very much for all of your help and support. It was really appreciated.

I would also like to express my appreciation to my ‘users’ Tore, Song, and Roy for taking time out of their own busy schedules to test the *ClockWorks* system and provide constructive feedback.

Thanks also to my fellow students at York University for all of your support and encouragement.

And finally, thanks to Ken for being my best friend through everything.

Table of Contents

List of Figures

Chapter 1: Introduction

The component-based style of programming has the potential to overcome problems commonly associated with the development of interactive computer software. However, component-based systems have generally been unable to realize this potential. Component-based programming involves building applications as frameworks of reusable software components [Nierstrasz *et al*. 1992]. As programs grow, these frameworks become complex and difficult to manipulate and understand. It is possible to overcome this difficulty of component-based programming by providing appropriate software development tools.

To demonstrate this point, a programming environment has been developed for Clock, a component-based system which supports the development of interactive software [Graham 1992]. The design of this environment was based on requirements identified through an evaluation and task analysis of the Clock system. The resulting program development environment, called *ClockWorks*, has been tested and shown to meet its requirements.

This chapter will first explain the difficulties associated with developing interactive software and how the features of component-based programming can overcome these problems. It will then explain the problems associated with component-based programming and discuss how these problems can be overcome. The final sections will explain the goals of this thesis and provide a summary of this document.

1.1 Interactive Software

Since the use of interactive software is widespread, attention must be given to supporting its development. Research has shown that software which is easy to use and learn is generally difficult to develop [Myers 1993].

A survey conducted by Brad Myers and Mary Beth Rosson [Myers and Rosson 1992] drew some conclusions about the average amount of code and development time spent on the user interface portion of the systems surveyed. Some of the results of this survey are summarized in the following tables. Figure 1.1 illustrates the percentage of resources such as time and source code which were taken by the user interface of the systems. Figure 1.2 shows what percentage of the systems surveys were developed with each kind of software development tool.

Resource	**Percent Taken by User Interface**
Code	45 %
Design time	45 %
Implementation time	50 %
Maintenance time	37 %

Figure 1.1 Percentage of software development resources taken by the user interface portion of applications. [Myers and Rosson 1992]

Development Tool	**Percentage of systems surveyed**
Toolkit	34 %
User Interface Management System	27 %
Interface Builder	14 %
No tools	26 %

Figure 1.2 Percentage of systems surveyed which used each kind of development tool. [Myers and Rosson 1992]

Common problems reported by the developers surveyed included:
- getting users' requirements
- writing help text
- achieving consistency
- learning how to use the tools
- getting acceptable performance
- communicating among various parts of the program

The results of this survey illustrate that the tools commonly used for the development of interactive software do not adequately meet the needs of developers.

This section will first discuss what makes interactive software difficult to develop. It will then go on to explain some factors which are required in order to successfully develop interactive software.

1.1.1 Problems Associated With The Development of Interactive Software

The difficulties associated with developing interactive software fall into two categories. These will be discussed in this section.

The first category includes problems associated with specific aspects of interactive software. In an interactive application, the user should control the order in which tasks are performed and should, therefore, not be forced to perform a series of required tasks in a specific order. The program should be able to communicate with the user through graphical representations which can display semantic feedback in response to user actions. Interactive software usually allows users to interact through input devices such as mice, pens, or touch screens. The concept of software development and the tools to support it have evolved to overcome these new requirements [Myers 1992]. New software development tools such as Toolkits, user interface development systems or interface builders have been used for the creation of interactive software.

The second category includes problems associated with specific tools meant to support interactive software development. Toolkits provide libraries of interaction techniques, routines, and classes which can be used to construct interactive software. Toolkits often contain so much information that they are difficult to learn and to use. User interface development systems are an attempt to integrate collections of tools, toolkits and libraries for the development of interactive software. The problem with such systems is that they are large, slow, difficult to develop and difficult to port between systems [Myers 1992]. Application or interface builders provide graphical editors through which the developer may specify the appearance of the user interface. While these tools are useful for allowing graphical specification of the interface, they limit the developer's choice of interface.

Knowing these difficulties will not necessarily help find a solution. A list of the requirements of interactive software is needed to have a useful perspective from which to propose a solution.

1.1.2 Requirements for the Development of Interactive Software

The goal of interactive software is to satisfy the needs of the user community by matching user tasks with the user interface of applications. Interactive software should be both easy to use and easy to learn. Strategies for program design and development have evolved in response to these new usability requirements. Programs which must satisfy the needs of a community of users cannot be defined, designed and developed in the traditional linear fashion because of the difficulty in determining the needs of the user community . In linear software development models, user testing does not occur until the program has been almost completely designed and implemented [Hartson and Hix 1989]. Modifications at this stage of development are very costly.

Developers of interactive software must include the users in the design phases, and users must be able to test the system as soon as possible after the initial interface design. In iterative design methodologies, the user community evaluates the design which is then modified and re-evaluated until the user community is satisfied. In some cases, a prototype of the system is built and tested with the users in the same fashion. In either case, the user community should be able to see what the system will look like and ensure that it will satisfy their needs before it is fully implemented. Many interactive applications are developed incrementally so that portions of the interface can be tested as they are implemented.

An additional complexity added to program design and implementation is the definition of the user interface portion of the code. This requires the ability to define graphical structures and direct manipulation style interaction.

There are also certain features which are now considered a standard part of any interactive application. These features include the ability to undo, abort, or cancel commands, and facilities for on-line help and information.

All of these factors contribute to the problem of developing usable, interactive applications which support the needs of their intended user community. The best way to ensure that such requirements are met is to provide development tools which address these issues.

Since interactive software has become an essential part of the computer software industry, development tools and methodologies need to be reorganized to support the requirements of this new style of software. Component-based software development is a new idea which is currently attempting to meet these requirements.

1.3 Component-Based Software Development

Component-based software development is a term which is loosely defined to include any system which involves building applications as a framework of previously defined software components [Nierstrasz 1992]. For the purpose of this thesis, the term will be defined more precisely. This definition is based upon a study and comparison of several different component-based systems. Several of these systems are discussed in Chapter 3.

1.3.1 An Explanation of Component-based Software Development

In component-based software development, applications are built as frameworks of software components which communicate with each other. These frameworks are usually called architectures.

A component is some functional part of an application. Each component has a specific purpose and can be defined with some interface through which it communicates with other components and the user. Depending upon the particular component-based system being used, components are used to define different aspects of applications such as interaction techniques (buttons, menus etc.) or data structures.

For example, a component could be used to represent a button. The definition of this component would define its appearance as well as its interface with other components (see Figure 1.3). The button receives the message that it has been pressed, and in response, it tells its state component to change and sends a message that it has been pressed. This message would trigger the action associated with the button.

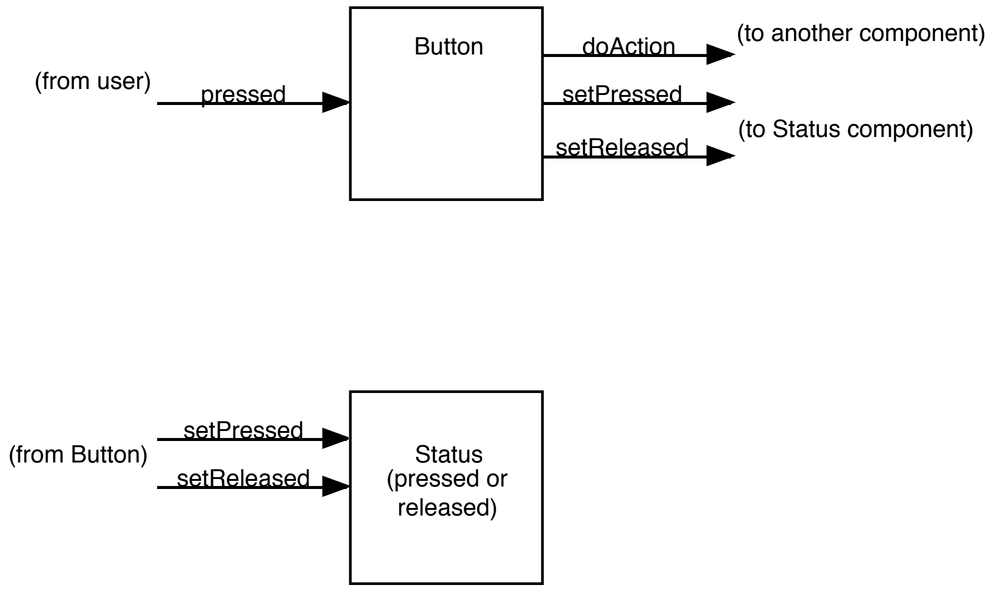


Figure 1.3 A Button component and Status component. The Status component holds the state of the button. Messages are represented as arrows.

A hierarchy of components can be used to define a more complex part of an application. For example, a dialog box or screen might be defined a collection of components each representing a different functional part of the user interface. In this way, complicated components are defined in terms several more specific components. The developer can define each component separately and then join them in a hierarchy to build an application.

Component-based programming systems generally provide some architectural model which is the basis for how components are structured to create applications. The construction of an application consists of defining components and connecting them to form an architecture. Applications can be created incrementally by defining and testing each component separately before connecting them.

Libraries of previously-defined components can be maintained so that applications can be constructed without the need to define every component from scratch. One of the main advantages of component-based programming is its ability to support software reuse.

1.3.2 How Component-based Programming Can Overcome Problems Associated with Interactive Software Development

Component-based programming has the potential to overcome many of the problems associated with the development of interactive software.

Each component responds to its own events and triggers events when necessary. Input from the user is handled directly by the component which is defined to accept that input. Therefore, the user of an application controls the order in which tasks are performed. The application can handle input events in any order.

One way in which a component can respond to input from the user is by changing its state or appearance. A component can also tell other components to change their state or appearance. This facilitates the implementation of semantic feedback.

Component-based programming is ideal for supporting the major requirement of the developers of interactive software - the need to support the tasks of their users. Components can be implemented and tested outside the context of the entire application. With this feature, interactive software can be developed incrementally. Modifications can be made before the entire application has been built.

Component-based programming systems are ideal for rapid prototyping. Developers can quickly define simple versions of their components, or small parts of their applications which can be executed and tested with users. Applications can be easily modified by swapping components or modifying their definitions.

Component-based programming is modular. Components can be replaced, removed, or modified with very little impact on the rest of the component hierarchy.

Component-based programming systems have features which make them ideal for developing interactive applications. However, without adequate support for managing the complexity of the component structures which define applications, component-based programming systems are too difficult to use. This prevents them from realizing their potential.

1.3.3 Problems Currently Associated With Component-Based Programming

The most important part of any application built with a component-based programming system is the architecture structure. This structure defines the way in which the application’s components are connected and the way in which they communicate with each other. The benefits listed in the previous section depend on the ability of the software developer to understand and manipulate this structure.

The architecture structure of a component-based application models a large amount of information. As applications grow larger, this structure becomes larger and more complex. The problem with component-based programming is that an application does not have to be very large before its architecture structure is difficult to understand and manage. When this happens, the developer will have a hard time manipulating the program.

If component-based programming systems do not provide programming tools which help developers manage the complexity of their program's architecture structures, the benefits of this style of programming are lost.

One of the main goals of component-based programming is to take advantage of its potential to support software reuse. In order to attain this goal, component-based programming systems must provide adequate tool support for developers:

> "effective reuse of software presupposes the existence of tools to support the organization and retrieval of components according to application requirements, and the interactive construction of running applications from components ... since the design of reusable frameworks is an iterative, evolutionary process, it is necessary to manage software and software information in such a way that designs and implementations can evolve gracefully." [Nierstrasz 1992]

It is believed that component-based programming has the potential to be an ideal method for building interactive applications and that this potential can be realized by providing the appropriate development tools.

1.4. Goals

The goal of this thesis was to define a software development environment to help developers of component-based applications manage the complexity of their program architecture structures. The *ClockWorks* is a tool which allows developers to build and manipulate the architecture structures which define Clock applications. The first version of *ClockWorks* has been implemented and tested with its users. Conclusions have shown that it has managed to overcome many of the problems which were associated with the ability to manage program architecture structures.

Although *ClockWorks* was developed specifically for the Clock system, the techniques and ideas used for its design and implementation are applicable to other component based systems. The similarities between Clock and other component-based programming systems will be discussed in chapter 3.

Care has been taken to ensure that the environment meets the needs of its users. The Clock Methodology was designed specifically to support the development of interactive software with the Clock system. This methodology was used to develop *ClockWorks* for two reasons: first, to determine how useful this methodology is for creating effective interactive software, and second to gain a better understanding of this methodology. This understanding is important because it is the basis for all programs developed with the Clock system. *ClockWorks* supports part of this methodology as is explained in Chapter 2.

1.5. Outline

This section gives an overview of the chapters of this thesis as well as a summary of the various stages involved in this work.

Chapter 2 introduces the Clock system and the Clock program development methodology. Clock is a component-based programming system which defines a specific architectural model which is used to construct interactive software. Chapter 3 sets the context of this research by describing related work. Several component-based programming systems are briefly described in order to provide a basis for comparing them with Clock. This section also describes several software development tools which provided some ideas used in the development of *ClockWorks*.

Chapter 4 describes how the requirements for *ClockWorks* were defined. An evaluation of the existing Clock system and a detailed task analysis of its developers was conducted. The results illustrated that many problems with Clock were associated with developer's inability to understand and work with program architecture structures. Chapter 5 describes how *ClockWorks* was designed specifically to support the requirements identified in chapter four. Chapter 6 describes some of the internal details of the implementation of *ClockWorks*. Chapter 7 presents the results of user testing and evaluation which took place throughout the development of *ClockWorks*. Chapter 8 draws conclusions for this thesis, explains its implications, and describes future plans for the programming environment of the Clock system.

Chapter 2: A Description of Clock

Clock is a component-based programming system supported by a methodology for the development of interactive applications. There are currently many obstacles which prevent Clock from being a successful system for creating interactive applications. However, it is still in the development stages and has the potential to be an effective system.

The current version of Clock is a prototype. To create an application, developers declare and define each component class to be used. The program's architecture is then defined in the Clock architecture language. Components in Clock are organized according to an architectural model, and defined in a functional programming language.

There are no interactive tools through which developers can create Clock programs. All components are declared and defined in text files. The architecture structure is also defined in a text file. The Clock compiler reads these files to create an application. A tool called Clock View can be used to read the architecture file and display it graphically. This tool was designed to support the documentation of Clock programs and not as a development tool.

Without interactive development tools, the Clock system is difficult to use and to learn. Architecture structures are difficult to understand and manipulate. *ClockWorks* has been designed specifically to overcome these problems.

The Clock methodology is well understood and based upon other successful interactive software development methodologies [Hartson and Hix 1989]. *ClockWorks* was designed using this methodology for two reasons: first, to prove that successful interactive applications can be built with the Clock methodology, and second to gain experience with the methodology so that *ClockWorks* can better support the tasks of its users.

In this chapter, the development of an Interactive Calendar program will be explained in terms of the Clock program development methodology. In this way, both the Clock system and its methodology will be clearly illustrated.

2.1 Clock Methodology

Figure 2.1 illustrates the structure of the Clock development methodology.

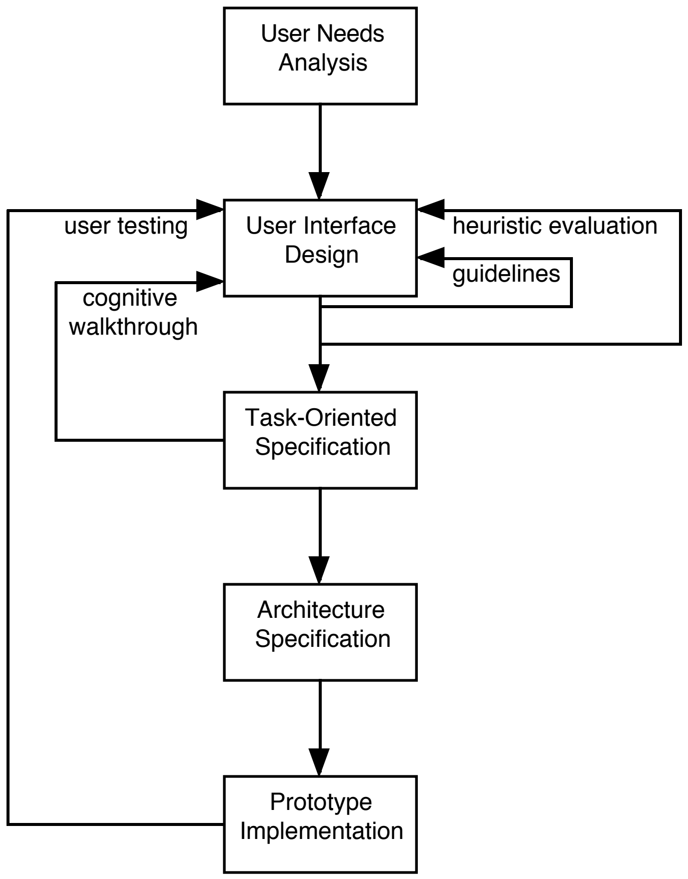


Figure 2.1 The Clock Methodology. The steps indicated in this diagram are performed sequentially, however, arrows indicate iterative loops where the results of user testing could lead to modifications to the program's user interface design.

The methodology begins with user needs analysis to identify the type of application the users require. The user interface of the application is designed from these requirements. This design is the most important stage in the Clock methodology as the user interface is the most important part of any interactive program. The users must be able to learn and use the application. Their satisfaction depends upon the quality of the user interface design. It is for this reason that all the iterative loops in this methodology result in modifications to the User Interface Design.

The task oriented specification is the stage in which the user interface design is formally evaluated through a cognitive walkthrough. Once the user interface design is in an acceptable state, the architecture of the application can be defined. It is this stage of the methodology which will be supported by the Clock program development methodology.

It is possible to combine architecture specification with prototype implementation in the Clock system. The application can be defined incrementally. Throughout the implementation, the application can be tested and the design may be modified in response to this testing.

The following sections will explain each phase of this methodology in more detail by illustrating the development of an Interactive Calendar program.

2.2 User Needs Analysis

The first phase in the Clock methodology is the User Needs Analysis. The goal of this phase is to become familiar with the users, their environment, and their tasks. The most important part of this phase is the task analysis.

The task analysis breaks the user's tasks into a hierarchical structure of goals, sub-goals and individual tasks [Carey *et al*. 1989]. The hierarchical task analysis for the Interactive Calendar is illustrated in Figure 2.2.

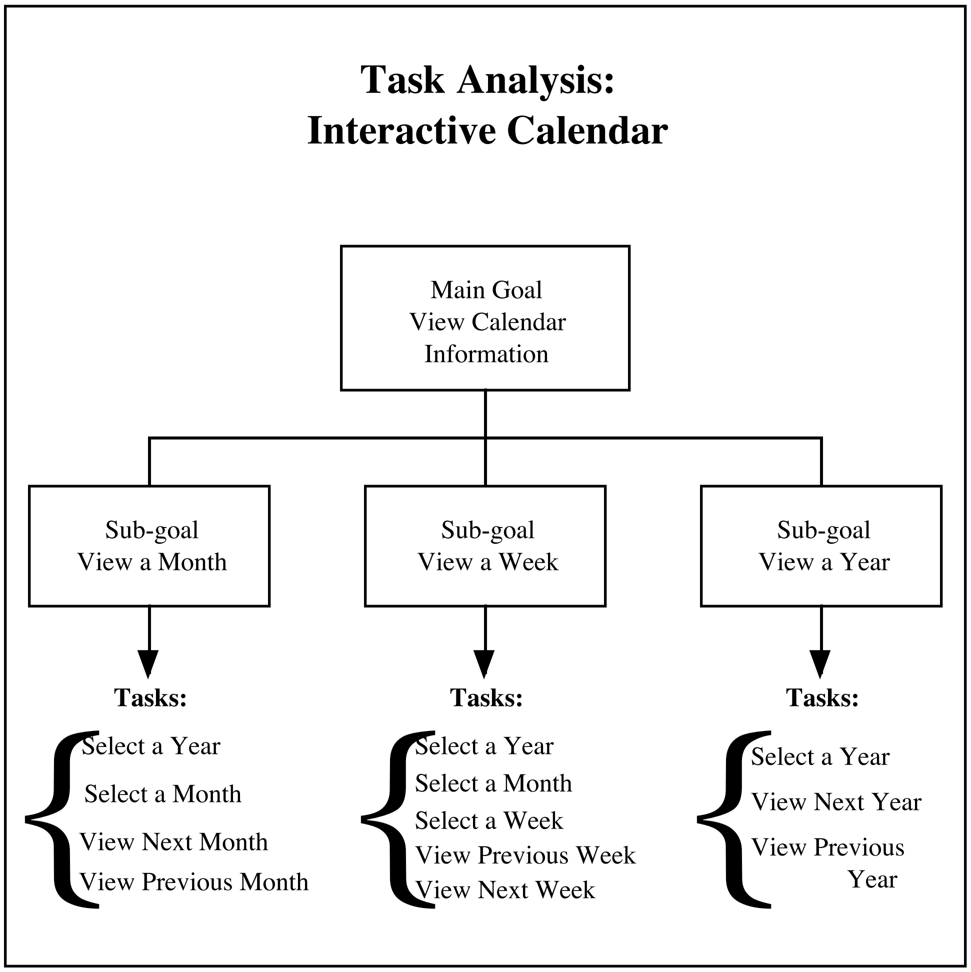


Figure 2.2 Hierarchical task analysis for the Interactive Calendar. The lowest level of this hierarchy defines individual tasks performed by the user.

Results from this task analysis are used to define the requirements of the application being developed. Each task is evaluated to document information which may be relevant to the system being developed. Relevant information might include the frequency at which this task is performed, or current problems associated with this task [Booth, 1989]. The tasks analysis is conducted by the developer and the user community to ensure that it is correct and complete.

2.3 User Interface design

The user interface design is based on the requirements defined in the user needs analysis. This design should be represented so that the users can understand what the proposed system will look like and how it will be used. They have the opportunity to comment on the design and participate in its fine-tuning. This process is called participatory design. Modifications required by the users can be made easily and inexpensively because no implementation has begun.

The user interface can be designed and evaluated according to user interface guidelines. These guidelines provide information about how user interfaces should be designed.

The User Interface design of the Interactive Calendar was sketched using a computerized drawing program. The design sketch of the month view screen is shown in figure 2.3. Since it was a fairly simple application, these drawings along with some very simple explanations were fairly self-explanatory.

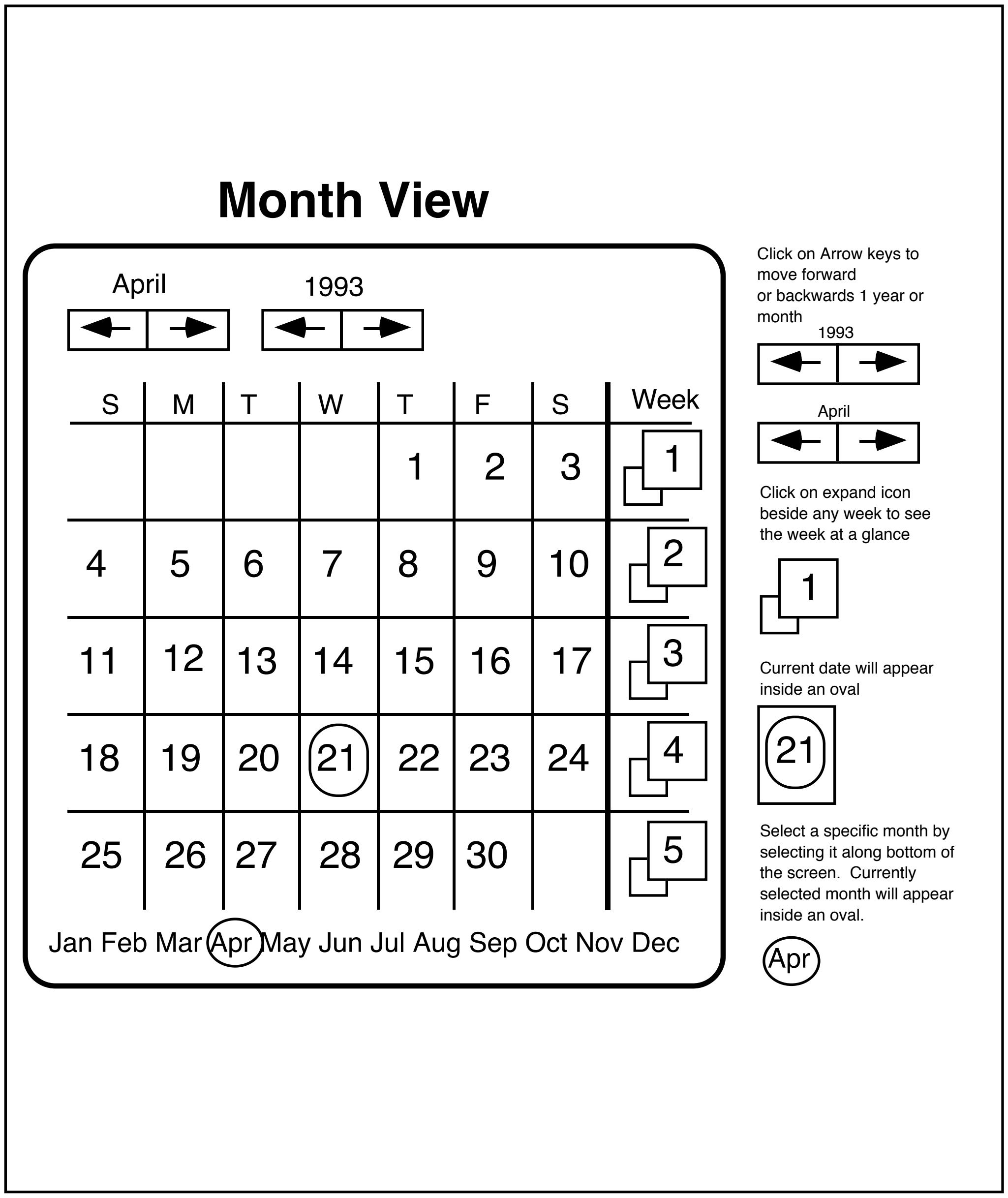

Month View
April
1993
S M T W T F S Week
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
Click on Arrow keys to move forward or backwards 1 year or month
1993
April
Click on expand icon beside any week to see the week at a glance
Current date will appear inside an oval
Select a specific month by selecting it along bottom of the screen. Currently selected month will appear inside an oval.
Apr

Figure 2.3 Design sketch of the month view screen of the interactive calendar program. The complete user interface design for this application included sketches for the week and year view screens as well.

2.4 Task oriented specification

The task oriented specification is a detailed, methodical evaluation of the user interface design which is performed from the perspective of the user.

The task oriented specification makes use of a language for representing user interactions with the computer. This language is called User Action Notation (UAN) [Hartson *et al*. 1990]. Every goal, sub-goal and task identified as part of the User Needs Analysis is represented in this language. The UAN description of the task "View a Month" from the hierarchical task analysis of the Interactive Calendar is shown in figures 2.4 and 2.5.

Tasks in UAN are represented in a chart with several columns. The first column lists the user actions using special symbols. User actions are written in the order they are to be performed. Unless otherwise specified, each task should be performed to completion before the next one is begun. Special UAN symbols have been defined to indicate more complex sequencing of tasks, such as tasks that can interrupt each other. The second column explains the semantic feedback provided by the interface in response to a specific user action.

Task: View Calendar Information

User Action	**Interface Feedback**	**Interface State**
(View (year) OR		
View (month) OR		
View (week))*		

Figure 2.4. UAN Description of task View Calendar Information *This task represents the main goal at the highest level of the hierarchical task analysis. The tasks are separated by OR because any one of the tasks may be performed, they do not need to be performed in order. The * symbol indicates that the tasks in the parentheses may be repeated any number of times. There is no information in the other columns because the detail of these tasks would be expanded elsewhere. The values year, month, and week are parameters of the tasks and represent the year, month or week the user would like to view.*

Task: View (month)

User Action	**Interface Feedback**	**Interface State**
if Screen=YEAR		

SelectMonth(month)	MonthScreen(month)	Screen = MONTH
		CurrentMonth = month

end if		
if Screen=WEEK		
~[GoBack] t	[GoBack]!	
s	[GoBack]-! MonthScreen(CurrentMonth)	Screen = MONTH
ShuffleMonth(month)		
end if		
if Screen = MONTH		
ShuffleMonth(month)		
end if		
(ViewNextMonth OR		
ViewPrevMonth)*		

Figure 2.5 UAN description of task View(month) *The* t *symbol represents the task of pressing the mouse button while the* s *symbol represents the task of releasing the mouse button. Since the user can be in any one of the three screens defined as part of the Interactive Calendar, an interface state variable called Screen will always hold the name of the currently active screen (YEAR, MONTH, or WEEK). The state variable* CurrentMonth *holds the value of the currently selected month. These state variables are set in response to user actions. This task is described from the perspective of either of the three screens. In the interface feedback column, the Macro* MonthScreen(month) *indicates that at this point, the month screen will be activated to display the specified month. The tasks* ShuffleMonth(month), ViewNextMonth, ViewPrevMonth *and* SelectMonth *would each have their own UAN descriptions.*

The process of creating a complete UAN description of a user interface is an excellent way of evaluating the user interface for consistency, completeness, and usability. The UAN description matches the hierarchical task analysis. In this way, the UAN will clearly identify any tasks which are not supported by the user interface. The UAN can also identify inconsistencies in the interface design which may confuse the user and make the system more difficult to learn or use. The UAN might also expose errors or gaps in the user interface.

This evaluation, along with the comments from the users will allow a developer to create a good, and acceptable user interface. There should be no need to make major modifications to the design once implementation has begun. Once the user interface is acceptable to the users, implementation can begin. The next two stages of this methodology are specific to the Clock system.

The task-oriented specification is the last part of the Clock methodology which is done from the perspective of the user of the system being built. Before the developer can go on with implementation, a transition must be made from the user perspective to the developer perspective. The UAN leads naturally to this transition. The detailed information about the user interface state which is included in the UAN can be used to begin the process of breaking the user interface design into components which can be connected to form a Clock Architecture. A rigorous approach to developing Clock architecture structures from UAN specifications was defined by Damker [Damker 1992].

2.5 Architecture design

The Clock Model describes how programs are broken down and constructed from components in the Clock system.

There are two types of components in Clock, event handlers and request handlers. Event handlers define some functional component in terms of its response to events, its appearance, and its communication with other components. A request handler holds a data structure and defines its initial value and its response to events.

A program is represented as a hierarchical tree built from components. Each node in the tree is a single event handler with a any number of request handlers attached to it. The architecture definition for the month view screen of the interactive calendar is shown in Figure 2.6. A graphical representation of this structure is shown in Figure 2.7.

```
subview monthView =
eh monthView : MonthView
        subview monthShuffle =
        eh monthShuffle : Shuffle
                rh id : Id
                subview control =
                eh rightArrow : Button
                        rh id : Id
                        rh status : Status
                end eh
        end eh
        subview yearShuffle =
        eh yearShuffle : Shuffle
                rh id : Id
                subview control =
                eh leftArrow : Button
                        rh id : Id
                        rh status : Status
                end eh
        end eh
        subview weeks =
        eh weeks : Weeks
                rh id : Id
                subview days =
                eh days : Days
                        rh id : Id
                        rh status : Status
                end eh
                subview weekSelectors =
                eh weekSelectors : Button
                        rh id : Id
                        rh status : Status
                end eh
        end eh
        subview monthBar =
        eh monthBar : MonthBar
                subview month =
                eh month : Button
                        rh id : Id
                        rh status : Status
                end eh
        end eh
end eh
```

Figure 2.6 Architecture Description of the Month View screen. In the original Clock system, the architecture was defined in the Clock Architecture Language.

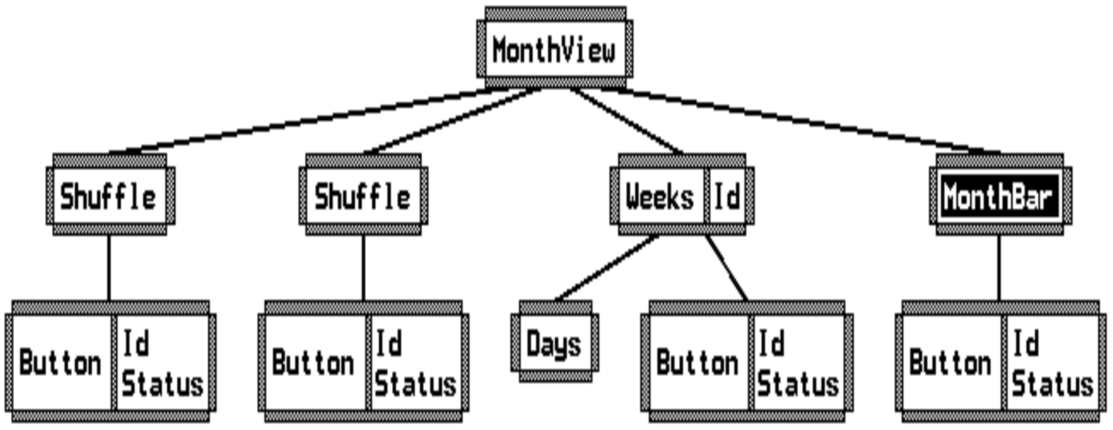


Figure 2.7 Graphical Representation of the Architecture Structure This picture is produced by the Clock View Tool. Request handlers are drawn to the right of event handlers. Nodes with no request handlers showing have been closed so that the request handlers are hidden from view.

The month view screen is made up of four different components which are the subviews of the *MonthView* event handler at the top level of this architecture.

The month bar allows the user to select the month by pressing a button. It creates one *Button* event handler for each month. The *MonthBar* event handler receives input from its children whenever one of them is pressed. It then sends an event to indicate which of its children was pressed so that the current month can be set accordingly.

The *Weeks* event handler creates the five weeks drawn on the screen to form a month. The *MonthView* event handler creates five instances of its weeks subview. Each week is made up of two subviews, its days and its week selector button. Each day displays its own date on the screen, the week selectors are Buttons which allow the user to switch to the week view screen. These components interact with a request handler which computes the date of a day given its week, month, and year. This request handler is attached to the parent of *MonthView*.

The last two subviews of the month view screen are both *Shuffle* components. The *Shuffle* component is used to move forward or backward through the years or months. It is made up of two *Buttons*, one causes the *Shuffle* to increase the current year or month while the other causes it to decrease the current year or month. The *Shuffle* interacts with request handlers which store the current year and month. These request handlers are attached to the parent of *MonthView*.

In this section of the architecture tree, there are two different request handlers. The *Id* stores a string which is used as an identifier. In cases where several instances of a subview are created, each one is passed a different value which makes them unique. This value is saved in the *Id* request handler for components that need access to their identifier. In this case, any component which draws a label on the screen, saves this label in its *Id* request handler. The *Status* request handler is part of the definition of a button. It holds a Boolean value to indicate whether the button is in a pressed or released state.

A tool called *Clock View*, is able to read a file containing the architecture description (see Figure 2.4). It can then display the tree graphically (figure 2.5). This tool was designed to assist in the documentation and explanation of Clock programs rather than to assist developers. It does not allow the developer to modify the architecture of the program.

There are three types of events which are used to allow components to communicate with each other. Inputs come either from the user (mouse or keyboard input) or they are passed from one event handler component to another. Requests are sent by event handlers in order to retrieve a value from a request handler. Updates are sent from event handlers in order to modify the value stored in a request handler. Updates can also be sent to event handlers as inputs. The complete architectural detail for a Shuffle component is shown in figure 2.8.

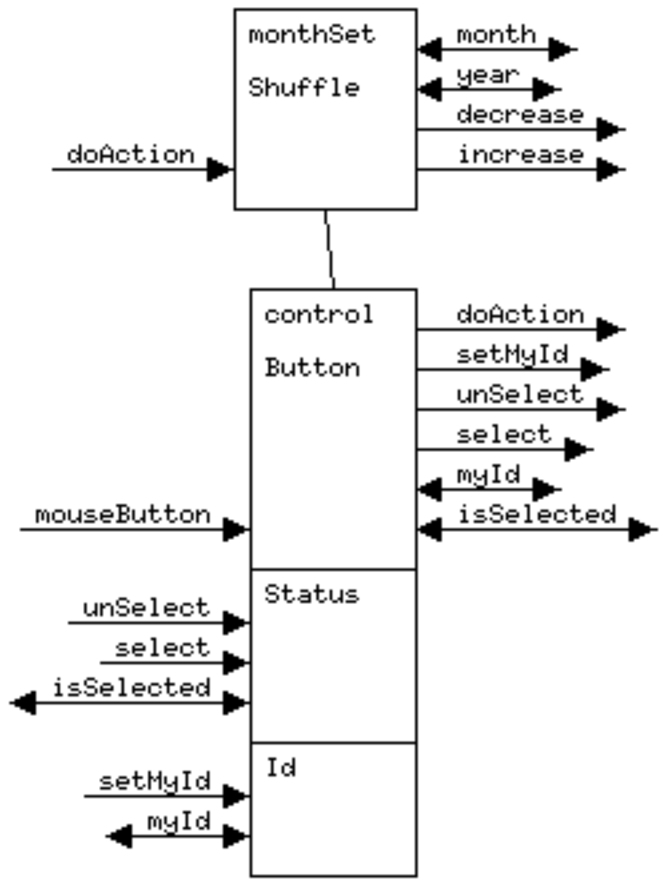


Figure 2.8 Full detail of the Shuffle component. In this representation, the inputs, requests and updates are drawn as arrows going into or coming out of the other components. Arrows on the left side are handled by the component, and arrows on the right side are used or sent by the component. An arrow with 2 heads is a request, an arrow with a single head is an update or input. Inputs are simply updates which are handled by event handlers.

Most of the requests and updates are used within a component group, allowing an event handler to interact with its own request handlers. Each arrow is drawn once from the component which makes the input request or update, and once into each component that receives it.

The *Status* request handler takes the request *isSelected*. This request is sent by an event handler to determine whether it should draw itself in a selected or unselected state. The *select* and *unSelect* updates handled by the *Status* request handler set its value to *selected* or *unSelected*.

The *Id* request handler takes the request *myId* which asks for its value. This is used by the event handler components to draw their label on the screen as well as to identify themselves to other components. The *setMyId* update handled by the *Id* request handler is called when the event handler is created and is used to set the *Id* to the value passed to the subview when it is created.

The remaining inputs, updates and requests are used to define the way the components interact with each other. The *Button* event handler takes an input from the mouse button and as a result makes the update *doAction*. When this update is called, the button sends the value of its *Id* as an argument. This update triggers the function which is associated with the button. The *doAction* update sent by each button is received as an input by the *Shuffle* event handler. In response to this input, the *Shuffle* sends either an *increase* or *decrease* update. The choice depends on the *Id* which is sent as a parameter to the *doAction* input .

When the *Shuffle* component makes an *increase* or *decrease* update, it sends its *Id* along with it. These updates are handled by request handlers which store the current year and month. Depending on the *Id* sent from the *Shuffle* component as part of the *increase* or *decrease* updates, either of these request handlers will change its value. Clock will then update any component which makes requests to these values. These components are re-drawn at this time.

Components in the architecture tree are instances of component classes. Each class has a declaration. Figure 2.9 shows the declaration file for the *Button* event handler. Figure 2.10 shows the declaration file for the *Id* request handler.

Each individual input, request, and update is defined as a function in the Clock language. Each of these functions should have a type signature to indicate the type of its arguments and the type returned by the function. Figure 2.11 shows the type signatures for several updates and requests.

A Clock architecture specification consists of a description of the architecture structure, a declaration for each component class, as well as a type signature for each input, update, and request. Once the architecture design is complete, each component can be defined in the Clock Language.

```
% class declaration: Event handler Button
ehClass Button
        inputs mouseButton
        requests myId, isSelected
        updates setMyId, select, unSelect, doAction

% class declaration: Event handler Shuffle
ehClass Shuffle
        subviews leftArrow, rightArrow
        inputs doAction
        requests myId, year, week, month
        updates increase, decrease, setMyId
```

Figure 2.9 Button event handler declaration. Each line in the declaration contains a list of names. Subviews are the names of an event handler's children. Each of these children can be instantiated with event handlers of any class. The last lines are the names of the inputs accepted by the class and the requests and updates sent by this component.

```
% Class declaration: request handler Id
rhClass Id
        requests myId
        updates setMyId

% Class declaration: request handler Status
rhClass Status
        requests isSelected
        updates select, unSelect
```

Figure 2.10 Id request handler declaration. Each request handler class declaration lists the names of the requests and updates which it handles.

```
request myId :: String
request isSelected :: Bool

update select :: UpdateEvent
update unSelect :: UpdateEvent
update setMyId :: String -> UpdateEvent
```

Figure 2.11 Request and update type signatures. All inputs and updates are declared as updates. They always return the type UpdateEvent *which is pre-defined in Clock. The updates* select *and* unSelect *have no arguments, but the update* setMyId *takes one argument of type String. These type declarations are generally included with the declarations for each class.*

2.6 Implementation

The fourth phase in the Clock methodology is the implementation (see figure 2.1). Once the architectural model of a Clock application or some part of a Clock application is complete, the implementation is fairly simple. Each component class is simply defined in the Clock language. Clock is a functional language, so each component's definition consists of a collection of functions. Each input, request, or update received by a component is handled by a function. Figure 2.12 shows the event function which responds to the *doAction* update received by the *Shuffle* component.

The *initially* function defines any actions to be done when the component is created. The *invariant* function is included so that the appearance of a specific component is always consistent with its state. The *invariant* function usually depends upon state values for the components which are stored in request handlers. The *invariant* function is always true, so when the value of a request handler is changed, the *invariant* function may change the appearance of the component. The *initially* and *invariant* functions for the *Shuffle* component are shown in Figure 2.13.

The view function defines the appearance of the event handler component. The view function for the *Shuffle* component is shown in Figure 2.14.

A request handler component defines a function for each update and request that it handles. Since each request handler holds a data structure, they each have a state whose type is declared when the request handler is defined. Figure 2.15 and 2.16 show the definitions of the *Id* and *Status* request handlers.

A standard update function takes the request handler's state as a parameter and returns the new state. Other parameters are defined by the developer. The name of an update function is the name of the update it handles plus “Updt”. Therefore the function which handles *setMyId* is called *setMyIdUpdt*.

A standard request function takes the state as a parameter and returns whatever value was requested. The name of a request function is always the same as the request it handles plus “Req”. Therefore the function which handles the *myId* request is called *myIdReq*.

```
doActionUpdt "<<" =
        decrease myId.
doActionUpdt ">>" =
        increase myId.
doActionUpdt _ =
        noUpdate.
```

Figure 2.12 Event function for the doAction *update handled by the* Shuffle *component. The definition of this function has three equations, each corresponding to a different argument. The third equation acts as a default which is called if the* doAction *function cannot match its argument to the patterns in the first two equations. If* doAction *is called with the argument ‘>>‘, it then calls the update* increase*, if the argument is ‘<<‘ then the* decrease *update is called.*

```
initially n =
        setMyId n.
invariant =
        noUpdate.
```

Figure 2.13 The initially *and* invariant *functions of the* Shuffle *component. The argument to the* initially *function (here called* n*) is passed to the component when it is created. On creation, the* Shuffle *component tells its* Id *request handler to set its* Id *to* n. *The* invariant *function is not used in the* Shuffle *component, so it calls the pre-defined update* noUpdate *which does nothing.*

```
theLabel =
        if streq myId "Year" then
                NumText year
        elsif streq myId "Week" then
                Text "Week"
        else
                Text month
        end if.

view =
        At origin stretching (
                above [
                        beside [
                                space 20,
                                Font veryLargeBoldItalicText (
                                        theLabel
                                )
                        ],
                        beside [
                                control "<<",
                                control ">>"
                        ]
                ]
        ).
```

Figure 2.14 View function of the Shuffle *component. In this example, the view function calls another function called* theLabel. *This function has been defined for convenience and it is not one of the functions required to define an event handler. This function creates the label which is part of the* Shuffle *component. The* Id *of the* Shuffle *component will either be "year" or "week". If it is "Year", then the label will be numeric and the pre-defined Clock function* NumText *will be used to create it. If the id is "Week", then the label is text and the pre-defined Clock function* Text *will be used to create it. The view function places a label above its subviews using a pre-defined Clock function* above. *The label places 20 spaces beside* theLabel . *The label is drawn using the font* veryLargeBoldItalicText. *The subviews are created by calling a function which is the name of the subview and sending it a particular argument. The* Shuffle *component has one subview (control). It creates two instances of this subview and puts them beside each other with the pre-defined Clock function* beside. *The entire component is drawn starting at the origin, its size is defined as stretching. This means that it will be drawn large enough to hold all of its graphical elements.*

```
% Request handler definition: Id
type State = String.

setMyIdUpdt _ t = t.

myIdReq s = s.

initially = "noId".
```

Figure 2.15 Clock definition of the Id *request handler. The* state *is the actual data structure stored in the request handler. The* Id *request handler holds a string identifier, so its type is* String. *The* setMyId *function takes a string, which is the value to set the* Id. *The update function does not use its first argument because its result does not depend on the current value of the request handler. It simply sets the request handler's state to the value of its second argument. The* myId *function returns the state of the request handler in response to the request. The initially function is called when a request handler is created. It has no parameters and simply returns the state of the request handler. In this case, the initial value of the* Id *request handler is set to "noId".*

```
% Request handler definition: Status
data State = Selected | NotSelected.

selectUpdt   _ = Selected.
unSelectUpdt _ = NotSelected.

isSelectedReq s =
        case s of
                Selected    -> True
                | NotSelected -> False
        end case.

initially = NotSelected.
```

Figure 2.16 Clock definition of the Status *request handler. The state of the* Status *request handler is a specially defined type which has only two values:* Selected *or* NotSelected. *The functions which handle the* select *and* unSelect *updates set the state to* Selected *or* UnSelected. *The function which handles the* isSelected *request, returns a Boolean value. It returns either* True *or* False *depending upon the value of the request handler. When the* Status *request handler is created, it is* NotSelected.

When a Clock application is run, the component classes are instantiated. Event handlers respond to inputs by making updates. Whenever an event handler makes an update, the update travels up the tree to the appropriate component. The function which responds to this update is triggered. The effects of the update then trickle back down the tree. Any event handler which makes a request to that request handler is re-computed. Its *view* and *invariant* functions are triggered.

Figure 2.17 shows the Interactive Calendar application running under the Clock system.

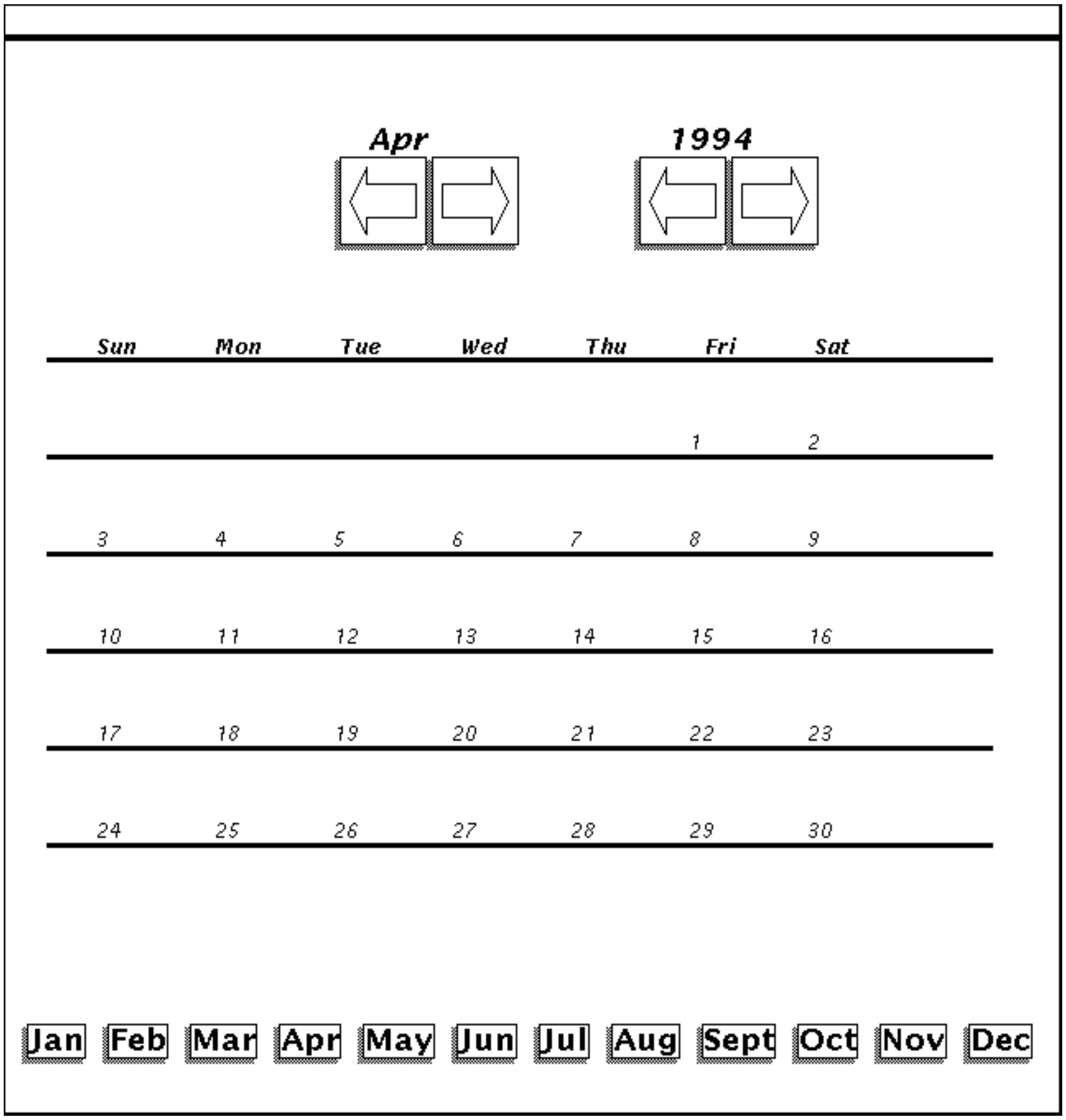


Figure 2.17 The month view screen of the Interactive Calendar Program.

When creating Clock applications, the developers generally work incrementally, creating individual components and then connecting them to form entire applications. Once the application has been implemented, it can be given to the users to be evaluated and tested. Several methods for conducting user testing were evaluated by Jeffries [Jeffries *et al*. 1991]. Apple computer publishes a set of guidelines for developing applications. These guidelines include a list of ten steps for setting up and conducting user observations [Gomell 1990]. Any of these methods may be used to test the user interface once it has been implemented. Results of these tests could lead to modifications to the user interface design and updates to the application.

2.7 Conclusion

The Clock methodology was designed specifically for the creation of Clock programs. The design of *ClockWorks* will follow the steps of this methodology to ensure that it meets the needs of its users.

The Clock system has the potential to make interactive applications easier to develop. Applications can be developed quickly and easily. They can also be modified by changing the components used to create the program's architecture. Libraries of reusable components can be designed and accessed by developers.

As will be shown in the next chapter, the current version of the Clock system has not proven to be very successful in its goal of making interactive applications easy to develop. This is mainly due to the lack of adequate tools to support the creation and modification of the architecture structure.

Chapter 3: Related Work

Although the *ClockWorks* system developed for this project was designed and implemented specifically for Clock, its scope is not limited to Clock. A study of several component-based programming systems has revealed that they all have similar requirements for their development tools. Section 3.1 will explain these component-based programming systems.

There are several program development tools whose goals are similar to those of *ClockWorks*. These systems are concerned with helping developers manage and understand complex software structures. These systems will be discussed in section 3.2.

3.1 Component-based Programming Systems

In this section, four different component-based programming systems will be described and compared with Clock in terms of their architecture structure and their development tools. The purpose of this section is to place the Clock system in the context of other component-based systems and to illustrate that ideas used in the development of *ClockWorks* would also apply to other component-based systems.

3.1.1 The Abstraction Link View Model

The Abstraction Link View (ALV) system is an architecture and programming method for building interactive, multi-user applications [Hill 1992]. Its architecture model is designed so that applications can separate shared application details from those which are unique for each user.

In ALV, an application is made up of three different kinds of components. Abstraction components hold abstract information such as data structures and the functions which manipulate them. View components define the graphical representation of an associated abstraction. A view component can be modified by the user through direct manipulation. The view components maintain a copy of any view-specific information which is stored in their associated abstraction components. Link components maintain consistency between the views and abstractions. The link components contain bundles of constraints which are evaluated each time the user manipulates the view or the abstraction component is modified.

Figure 3.1 illustrates how ALV is used to create an interactive application.

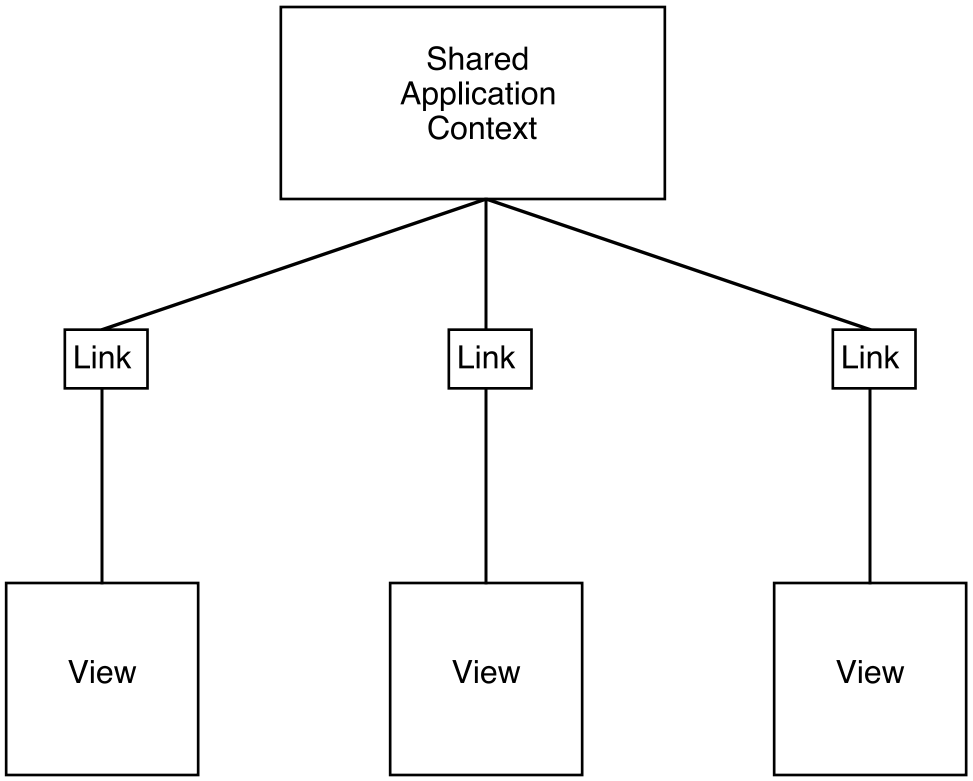


Figure 3.1 Model of an ALV application. [Hill 1992] An application usually has one abstraction component which is made up of several views. Since ALV is designed for multi-user applications, the abstraction is shared between all users, and there is generally one view for each user. Each view is connected to the application through its own link component.

Each view can be made up of a hierarchy of view components. A view component may maintain its own copy of information from the abstraction which it needs to compute its view. This allows a view component to modify its appearance without waiting for the associated event to be sent all the way to the abstraction.

The ALV model is similar to Clock in its structure. Currently, Clock cannot work with multi-user applications, but there are plans to update Clock for this purpose in the future.

Abstraction components are similar to request handlers in Clock. They hold and manipulate the program's data. Event handlers are similar to ALV view components. The main difference is that view components actually copy the data they need from the abstraction, while event handlers get their required data from request handlers. ALV defines a link component which manages communication between components, while Clock uses updates, requests and inputs for this purpose.

The ALV programming environment provides an editor and debugger for coding and running components. Future plans for ALV include a tool for non-programmers which would provide reusable components in a library. The developer could select abstraction and view components from these libraries and connect them by drawing the link as constraints between them.

3.1.2 Garnet

Garnet [Myers 1990, 1992] is a system designed to make user interfaces easier to build. Garnet has only one kind of component which is called an object. Objects are made up of slots which hold data and methods which manipulate the data.

Objects contain all the information required to define an interface component. Data slots may define the location, size, appearance, and state of the component. Initial values of slots may be set or inherited from a related object. Method slots perform functions related to the objects.

In Garnet, interaction between the user and objects is defined through interactors. Interactors, which define a specific style of interaction, can be attached to the object to give it a way to accept input.

Constraints are used to define the relationships between objects. Constraints may make the value in a data slot dependent upon a value in another data slot. The slots corresponding to the appearance of the component can be constrained to rely on values stored in other slots which correspond to the object's state. Figure 3.2 shows an example of how a button object could be structured in Garnet using a simplified syntax.

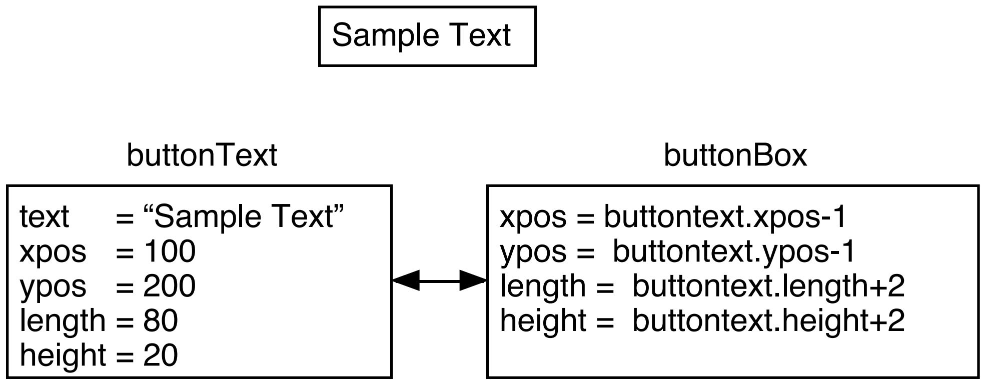


Figure 3.2 Button object and an illustration of how it could be modeled in Garnet. In this example, a Garnet button object is created from a text object and a box object. The dimensions of the box are constrained to the dimensions of the text so that the box will always be drawn around the text. To create a real button, an interactor would be attached to the button object. The button interactor would allow the object to accept input and trigger a callback function which would correspond to the action. Note that this example is used to explain how objects are modeled in Garnet. The example is very simplified and is not meant to be an example of actual Garnet syntax. This example was used in a university course on Interactive System Design to illustrate the idea behind Garnet, rather than its syntax.

Although Garnet is quite different from Clock and other component-based systems, it can be used in a similar fashion. In Garnet there is no architectural model upon which applications must be structured. Developers can use any architectural model they prefer. It is possible, for instance, to simulate the Clock architectural model in Garnet.

In Garnet, an event handler can be simulated by an object which will contain all the information required to define an interface component. Slots can be defined for each function which defines an event handler in Clock. In Garnet, a draw method is defined to compute the appearance of an object. This method is called by the graphics system whenever an object needs to be re-drawn. When the object is instantiated, it is drawn according to the data given in its slots. In Garnet, an aggregate can be used to group graphical objects into collections, so they can be instantiated much the same way as the

Clock views which are defined as a group of subviews. In both systems, these re-drawing functions are never called directly by the programmer.

The data slots of objects can be used to hold the information which is stored in request handlers in Clock. The value of a data slot may be retrieved by the name of the slot much like a variable's value can be obtained by its name. The value of a slot may be set or changed by using its name and assigning it a value much like a variable is assigned a value. Method slots hold functions or methods. These methods may be used to implement messages sent to objects. Data slots can simulate Clock request handlers as well as requests, and updates.

Garnet's constraint system can be used to make the value in a data slot dependent upon a value in another object's data slot. In this way, the slots corresponding to the appearance of the component can be constrained to rely on values stored in other slots which correspond to the state. This is similar to the Clock invariant function.

The Garnet system provides a number of tools which support the tasks involved with development of a Garnet program. There are two complete widget sets which contain pre-defined objects or collections of objects. An interface builder and a dialog box creator are provided to assist in the graphical layout of objects and object hierarchies. Constraints are managed with a spreadsheet tool. Garnet has a very elaborate programming environment, which includes a tool called Lapidary which generates components from direct manipulation descriptions of the user interface.

3.1.3 Model View Controller

The Model View Controller (MVC) paradigm is an architectural model for developing interactive software in Smalltalk [Krasner and Pope 1988]. An MVC application is made from three kinds of components: models, views and controllers. Model components contain application code and data representing the system state. Model components are made up of Smalltalk objects. View components maintain the screen display. Controller components interact with the user. Figure 3.3 illustrates how applications are structured in MVC.

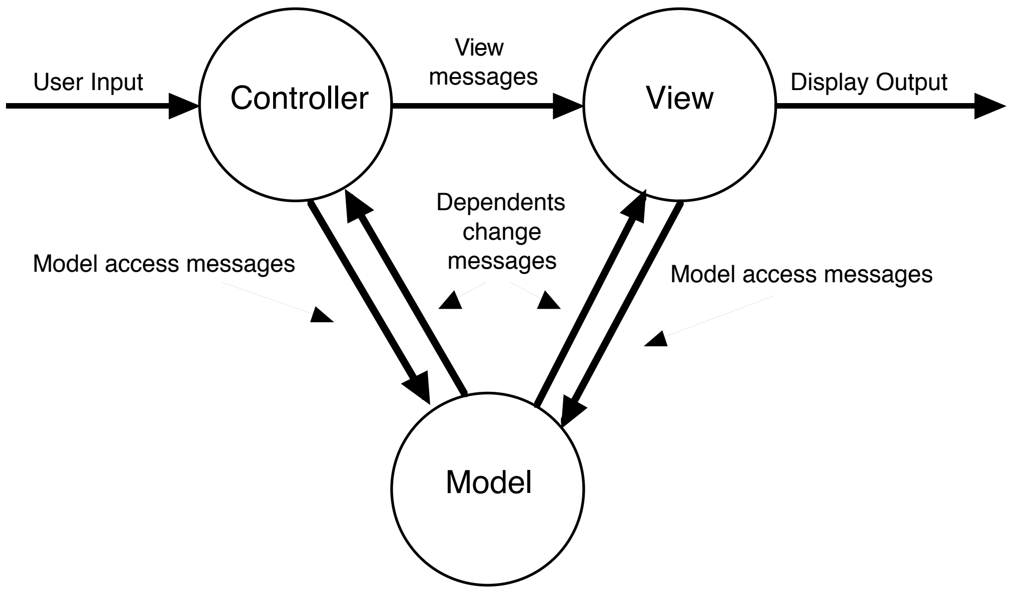


Figure 3.3 MVC Program Structure. In MVC, an application has a single Model component which can be made up of one or more view-controller pairs. Messages from the user are accepted by the controller component and passed either to the view or to the model. Changes to the model are broadcast to each view-controller pair.

In MVC, controller components handle input events like the request handlers in Clock, except that they pass the events on to the model component which stores the application's data structures and functions. The model component is similar to a collection of Clock event handlers without view functions. An MVC model component manages a collection of view components which define the way in which the various parts of the model are represented to the user.

The MVC environment provides several tools which assist in low level programming tasks. Workspaces, editors and a browser allow text manipulation. The inspector tool provides a list box which contains the names of values of variables associated with an object.

3.1.4 Presentation Abstraction Control

In the Presentation, Abstraction, Control (PAC) model, an interactive system is constructed as a hierarchy of interactive objects [Coutaz, 1987a, 1987b, 1989]. Interactive objects are made up of three different kinds of components. Presentation components define and display a graphical representation of the object. Abstraction components store data associated with the object. Control components maintain the link between Presentation and Abstraction components. Control components also maintain links between objects in the architecture.

Figure 3.4 illustrates the PAC architecture of a simple Pie chart which allows the user to change the size of the divisions by direct manipulation or by directly entering a numeric value.

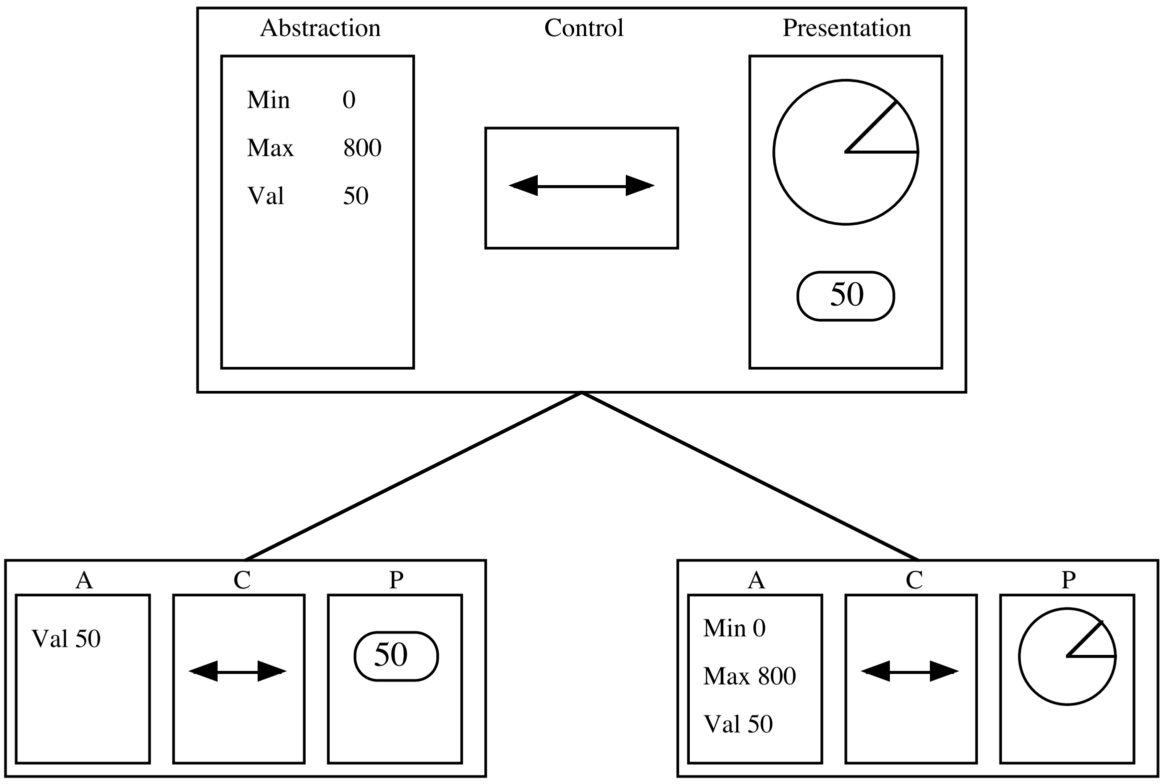


Figure 3.4 The PAC architecture structure of a Pie Chart. [Coutaz 1989] Each interactive object has a presentation component (P), an abstraction component(A) and a control component (c) The Root object in this tree represents the entire pie chart. Its detail is delegated to its children. The left child controls the text editor and the right child controls the pie chart. The control component of each object connects the abstraction to the presentation. Events from the user enter this structure from the bottom and are received by the controller components at the lowest levels. These controllers update their own information and then pass the event up to the control component at the next highest level. Events from the application would enter this structure from the top level and be handled by the control component at the highest level. This control component would then pass the event down the tree to the components which need it.

The PAC model is very similar to Clock. Abstractions are similar to request handlers, the Presentation components are similar to event handlers, and the control components are similar to the inputs, requests and updates used in Clock. The main difference is that in PAC data elements in abstraction components are duplicated in the tree, while in Clock, request handlers are placed at a position in the tree which allows them to be accessed by children at a lower level.

When the PAC model was being designed, plans for its programming environment included tools to edit and debug an application as well as graphical editors to allow the developer to specify the appearance of objects and the application. There was nothing in the research conducted for the PAC system to indicate that it had any plans to build development tools designed specifically to assist developers in creating and manipulating the architecture model.

3.1.5 Summary

In general, these systems are similar to Clock in that they provide a framework for structuring the functional components which make up an interactive system. The structures are hierarchical in nature and intuitively graphical, so these systems would benefit from a graphical method to represent the structure and allow direct manipulation by the developer. None of the systems studied provide tools which support this kind of representation and manipulation of the program's structure.

3.2 Software Development Tools

ClockWorks was designed to visually represent component architectures in the same way that developers draw and understand them. Developers should be able to work with pictures wherever pictures are the best way to express their ideas.

This section presents several software tools that share this focus with *ClockWorks*. Their goal is to provide graphical representation and manipulation of complex concepts such as high-level program design, or complex data structures.

3.2.1 Garden

The Garden system is an environment which supports conceptual programming [Reiss 1987]. Conceptual programming allows developers to build programs from the same conceptual models they use to understand them. These conceptual models are often graphical. For example, most developers draw a linked list data structure as a series of boxes connected by arrows. A tree is a series of boxes or circles connected in a hierarchical structure. Most programming styles or systems force developers to translate ideas from their conceptual model into something which can be represented in a text-based language. The goal of the Garden system is to make this translation easier by allowing developers to create graphical, high-level specifications of their conceptual models. These specifications are refined incrementally until an executable program has been defined.

The Garden approach is similar to that of *ClockWorks*. The architecture model of a component-based program is actually a conceptual model of the program's structure. In component-based systems, this model becomes a part of the finished application. It would therefore be very useful to have a programming environment in which this conceptual model could be represented and manipulated graphically.

When Garden was being created, plans for its implementation stressed the requirement that programmers be able to view complex or large program designs from various perspectives and at varying levels of detail. This was one of the major requirements of *ClockWorks*.

Additional plans for Garden include support for prototype evaluation and animated execution. A high-level program design created by Garden will be executable and a debugging tool will provide an animated view during program execution so that the program can be understood while it is running. These features will allow programmers to experiment with and modify their designs.

These features would also be useful to Clock developers. In Clock, a simple set of defaults can be designed for each type of component so that the architecture will be executable at all stages of its development. A graphical representation of the architecture of a Clock application could be used as the basis for a tool to animate the execution of a Clock program.

3.2.2 Software Landscape

The software Landscape is an environment for illustrating the relationships between program components [Penny 1993]. The term component has a broader definition in the context of the Landscape. It is the term used to describe any graphical software entity such as designs, modules, classes or versions.

The goal of the software landscape is to assist developers in understanding the structure of large scale software systems as they are being built or modified. The notation defined by the landscape programming tool defines how to connect these graphical software entities with non-graphical software entities such as source code. This connection is formally defined.

By using the Software Landscape, the developer is able to draw pictures representing the high level details and structures of large software systems. This assists developers in planning, understanding, and maintaining their software. A significant amount of the information required for software development is contained in easy to understand pictures which allow the developer to see varying levels of detail. Figures 3.5 and 3.6 illustrate example Landscape diagrams.

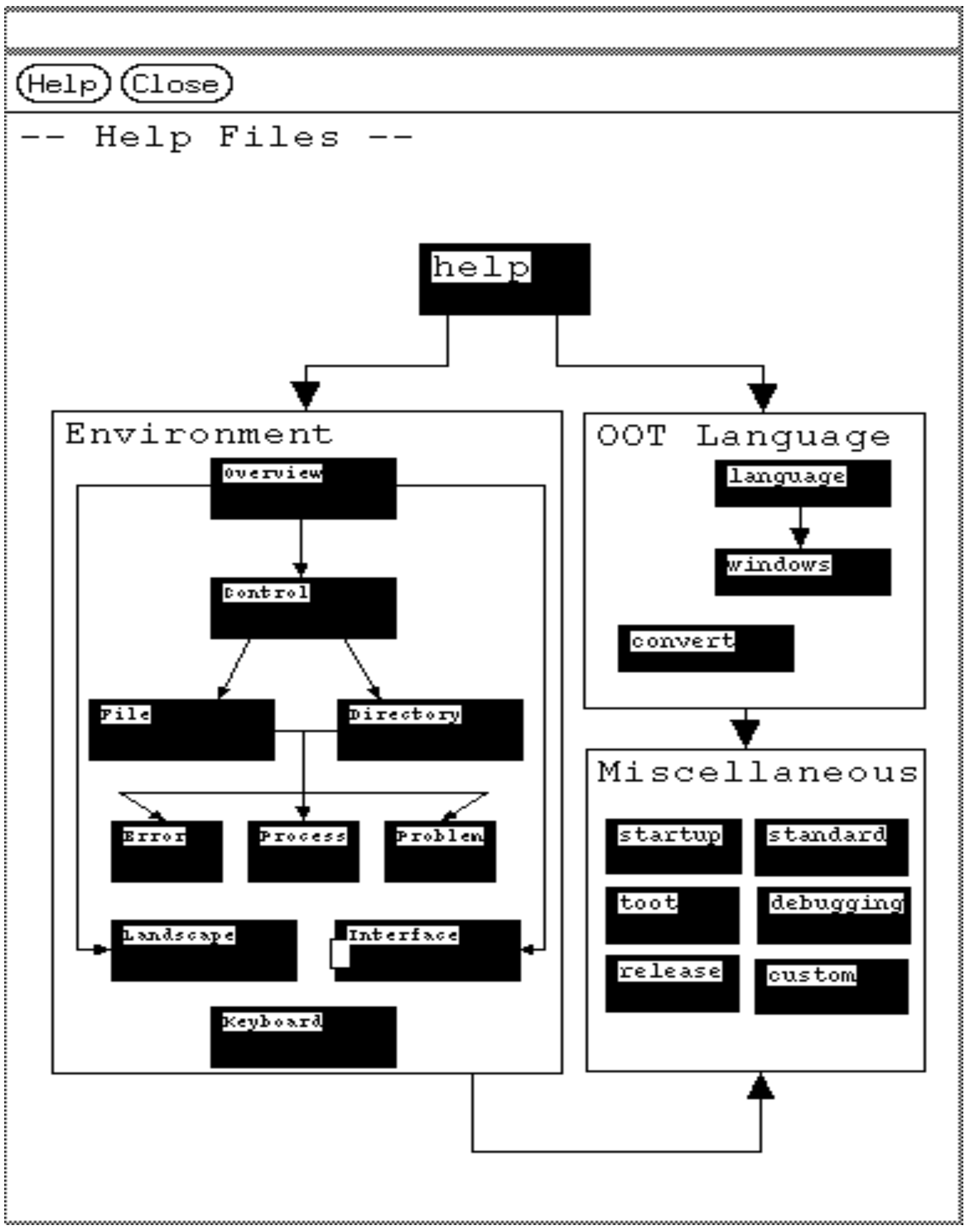


Figure 3.5 Screen capture of the landscape view of the Object Oriented Turing help system. This figure was taken from the Object Oriented Turing Environment [Mancoridis, Holt, Penny 1993]. Each object in this Landscape diagram is a help file.

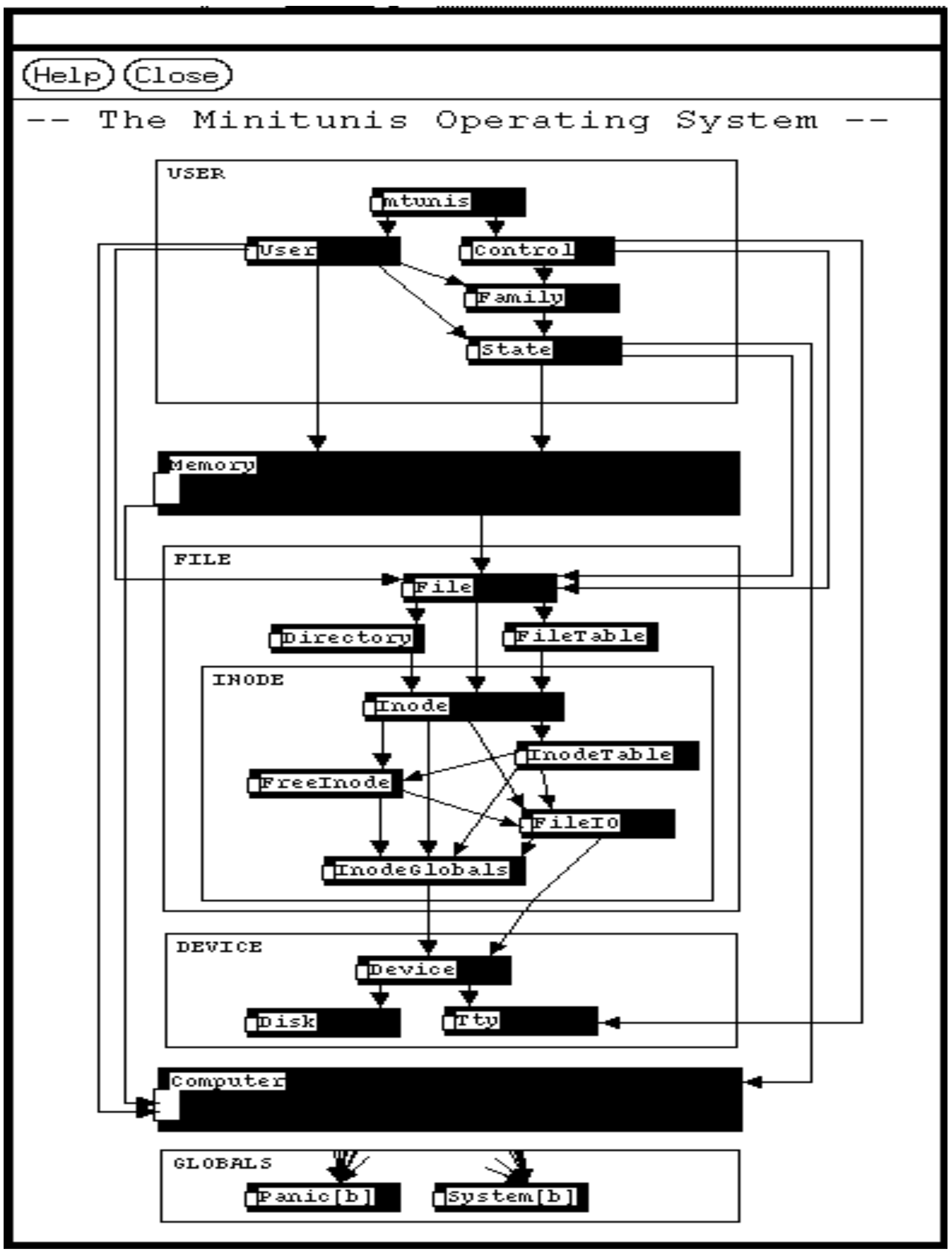


Figure 3.6 Screen capture of the landscape view of the MiniTunis operating system. This figure was taken from the Object Oriented Turing Environment [Mancoridis, Holt, Penny 1993]. Each object in this Landscape diagram is a Turing class.

The software Landscape creates a graphical representation any structure of components used to define any aspect of software development. It allows developers to create pictures of their program structures so that they can be easily understood and manipulated. Ideas from the Software Landscape can be used to design tools for creating, manipulating and understanding the architecture structure of component-based programs.

3.2.3 PegaSys

The PegaSys (Programming Environment for the Graphical Analysis of Systems) program development system was designed to replace and overcome problems associated with traditional program documentation techniques such as flowcharts and dataflow diagrams [Moriconi and Hare 1985]. The goal of PegaSys is to replace these traditional methods with an automated tool with which developers can work directly with pictures which represent program components and concepts. Using this tool, programmers could specify a high level program design as a series of components communicating and interacting with each other. These diagrams could be incrementally refined to the point where code could be associated with each component.

Both PegaSys and *ClockWorks* are an attempt to create representations of complex programs. These representations should be easy to follow and understand so that programs could be easily developed and maintained.

In PegaSys, a program is designed as a hierarchy of components. Components are represented by pictures. Each component can be refined to include more detail. These refinements are performed incrementally until the program's design is complete. Once design is complete, code is written and associated with the lowest level in the hierarchy.

As PegaSys was not built with component-based programming in mind, the resulting system is quite different from *ClockWorks*. The focus of research and development of PegaSys has been on creating a formally defined graphical language and a formally defined set of actions which verify the correctness of each refinement.

3.2.4 Summary

The programming systems described here all have goals similar to those of *ClockWorks*. Complex program structures should be represented in such a way that developers are able to understand and work with them. With this kind of support, developers will be better able to design and manage their software projects. Also, this style of programming tool can lead to more advanced programming support for prototype execution or animated execution views.

3.3 Conclusion

In understanding the various component based programming systems and comparing them to Clock, it becomes clear that all have certain similarities. In particular, each system defines some concept of a component. The components are attached to form a structure upon which an interactive application is built. *ClockWorks* is intended to help programmers manage the complexity of their program architectures in the Clock system. Since component based programming systems use an architectural model to structure components, ideas from *ClockWorks* apply to other component based systems.

The goal of *ClockWorks* is to help programmers understand and manipulate the architecture structure of their Clock programs. The program development systems studied share this goal.

The remaining chapters will describe how *ClockWorks* was designed, developed and tested.

Chapter 4: Identifying the Requirements of *ClockWorks*

This chapter will explain the methods used to identify the requirements of *ClockWorks*. It has already been explained (in Chapter 1) that component-based programming systems require tools to help developers manage the complexity of their program architecture structures. In this chapter, this requirement is refined to provide a more detailed list of requirements.

The first step in the Clock programming methodology (as described in Chapter 2) is the user needs analysis. Its goal is to identify the tasks of the users and define exactly what they require. Since the development of *ClockWorks* made use of the Clock methodology, the first step of its development was User Needs Analysis.

Section 4.1 explains the first stage of the identification of the needs of Clock users. This was a survey of a group of students who used Clock in a university-level course on user interface development. Their reactions helped to narrow down the problems users had learning and using the Clock system.

The user survey provided some interesting conclusions, but they were too general to lead to specific requirements. A detailed task analysis of the Clock programmer was conducted to gain a more precise understanding of the tasks involved in developing a Clock application as well as the problems commonly associated with these tasks. The results of this task analysis were reviewed by the Clock users and modified according to their comments. The task analysis is described in section 4.2.

The results of the task analysis led to the definition of the requirements of *ClockWorks*. Section 4.3 is a list of the ten requirements identified for *ClockWorks*.

4.1 User survey

A version of Clock was used as part of a fourth year university level course called Interactive System Design. The students used the Clock-Turing system. This is a version of Clock in which components are defined in Turing rather than in the Clock functional programming language. At the end of this course, a survey was conducted to determine how a group of users reacted to the Clock system. Its goal was to analyze Clock on the basis of how easy it was to learn and use. The results of this survey are summarized below. Section 4.1.1 explains the users comments about the Clock View tool while section 4.1.2 explains their reaction to the entire Clock system.

Eighteen students completed the survey. Of these, half had no previous experience programming interactive systems, and half had some experience with systems like

Macintosh, MS Windows and X. Those with experience felt that Clock was more difficult to learn and to use than these other systems.

4.1.1 Response to the Clock View Tool

The first section of the user survey was to determine the student's reaction to the Clock View tool. The Clock View tool (CV) is used to display Clock architectures. Its initial purpose was to create documentation for Clock programs. It interprets the Architecture file and displays the event handlers and request handlers as a tree. It does not display any information about requests, updates, and inputs. Figure 4.1 is an architecture diagram produced by the Clock View Tool.

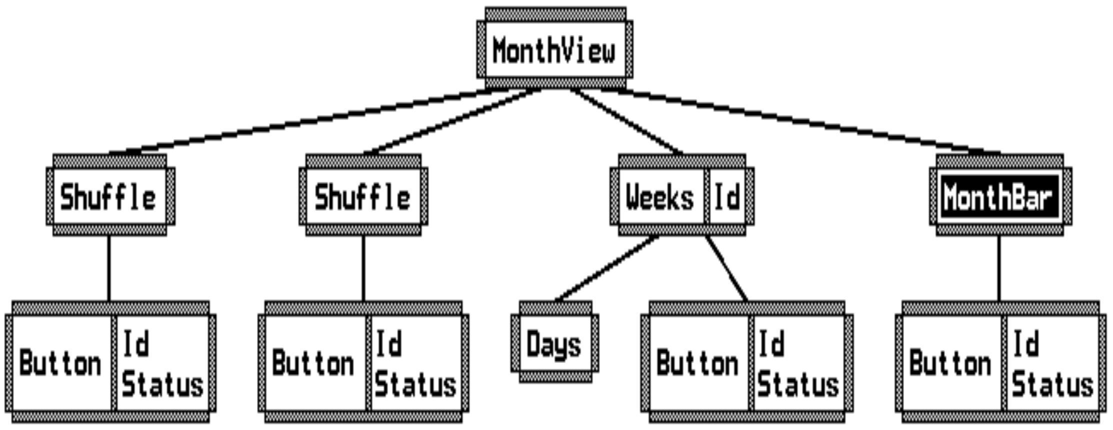


Figure 4.1 Architecture of the Month View screen of the Interactive Calendar. This picture was produced by the Clock View Tool.

Most of the users surveyed found the CV tool helpful. About half had some trouble getting started, but most found they had no trouble after they had used it for a short time.

With respect to the usefulness of CV, the users' reports were mixed. About half the students found the tool useful in designing their program's component structure, and understanding the flow of inputs, requests, and updates, while half reported that it was not useful. About half the students reported that CV did not display enough information about the component structure to be very useful, while half had the opposite opinion.

The problem with the CV tool is that it wasn't designed to be a programming tool and therefore, it does not do enough. The CV tool provides no verification of the architecture structure, and will display an incomplete or incorrect architecture as long as its definition is syntactically correct. Since CV does not display any inputs, requests, or updates, developers cannot use it to understand these important aspects of their architecture structure. Developers tend to move from the architecture design into implementation and then discover that their architecture was incorrect. The CV tool was intended only to view Clock architectures, so it cannot be used to manipulate them in any way. These problems become more evident in the rest of the survey.

The CV tool allows developers to view the general structure of event handlers and request handlers. The students' response to this tool indicates that it may have been useful if only it could support more of their tasks in creating and manipulating Clock architectures.

4.1.2 Response to the Clock System

This section will describe the results of the second part of the survey. Its goal was to identify the student's reaction to the Clock system in general.

Almost all users reported that they did not find the Clock system easy to use after some experience with it. Most agreed that implementation was frustrating, and that they had problems with the implementation even after the architecture design was complete.

These reports can lead to the conclusion that the Clock system does not have adequate support for program implementation. However, these problems all lead back to the architecture design. Given a complete and correct Clock architecture, implementation would involve the simple task of coding the functions which define each component. However, since the CV tool allows programmers to proceed to implementation with an incomplete or incorrect architecture design, the task of implementation also involves making corrections and modifications to the architecture design. This extra work could be eliminated if the tools supporting architecture design were more complete.

Several of the students commented on this problem when asked what they found to be the most difficult part of the Clock system.

> "Architecture specification seems to be the most critical part for implementation and I found it frustrating when the architecture "looks OK" but when we actually came to implementation and when changes are needed, we have to access a lot of files to make those changes..."
>
> "Architecture errors mostly not found until run-time, not in compilation. "
>
> "Why would one want to draw the tree structure by hand and then see the same on the screen? "
>
> "The transfer from Arch to implementation was not very clear at the beginning, at the end was obvious, a little too late."

When asked what they found most difficult about the Clock system, several students complained about implementation problems, one calling the implementation "a certain headache". Others reported having trouble understanding the relationships between event handler and request handler classes, and understanding the flow of inputs, updates and requests in the architecture tree.

There were many complaints about unclear feedback from the Clock system, incomprehensible error messages and lack of documentation. Some users also complained about slow and inefficient performance. These problems are caused by the current implementation of the Clock system, which is only a prototype. These problems will be addressed when the Clock system is updated. Therefore, they were not used for this project.

The user survey helped support the conclusion that the Clock system would be greatly improved if it had more tool support specifically geared towards managing the complexity of Clock architecture structures.

4.2 Task Analysis

The results of the survey were useful in identifying the root of the problems associated with the Clock system. However, its results were too general to be used to identify concrete, specific requirements for *ClockWorks*. In order to accomplish this definition of requirements, a detailed task analysis was conducted.

The Interactive Calendar program which was described in chapter 2 was implemented as a case study so that the task analysis could be carried out. Throughout its development, a detailed list of tasks was kept.

The list of tasks produced from this program was arranged in a hierarchy which began with a main goal of creating an interactive application This goal was broken into sub-goals which were then divided into the specific tasks performed to carry them out. Figure 4.2 illustrates the highest level of this hierarchy starting from the main goal.

Each sub-goal is one of the stages in the Clock methodology which was described in Chapter 2. The Architecture Design and Implementation sub-goals were expanded fully. Since *ClockWorks* only deals with these two stages of the Clock Methodology, the other sub-goals were not expanded. The full task hierarchy for the Architecture Design and Implementation sub-goals is shown in Figure 4.3 and 4.4 respectively. Appendix A contains the complete description for each task in these hierarchies.

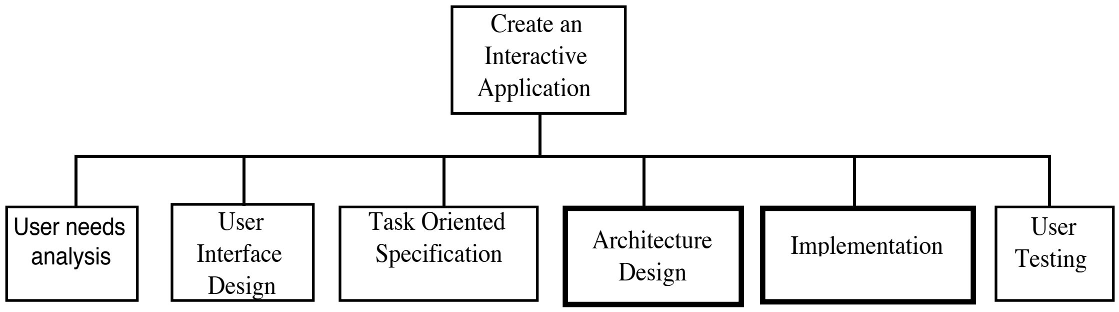


Figure 4.2 Highest level of the Hierarchical task analysis of the creation of interactive applications with Clock. This hierarchy begins with the main goal. This goal has then been broken down into sub-goals which correspond to the steps of the Clock methodology.

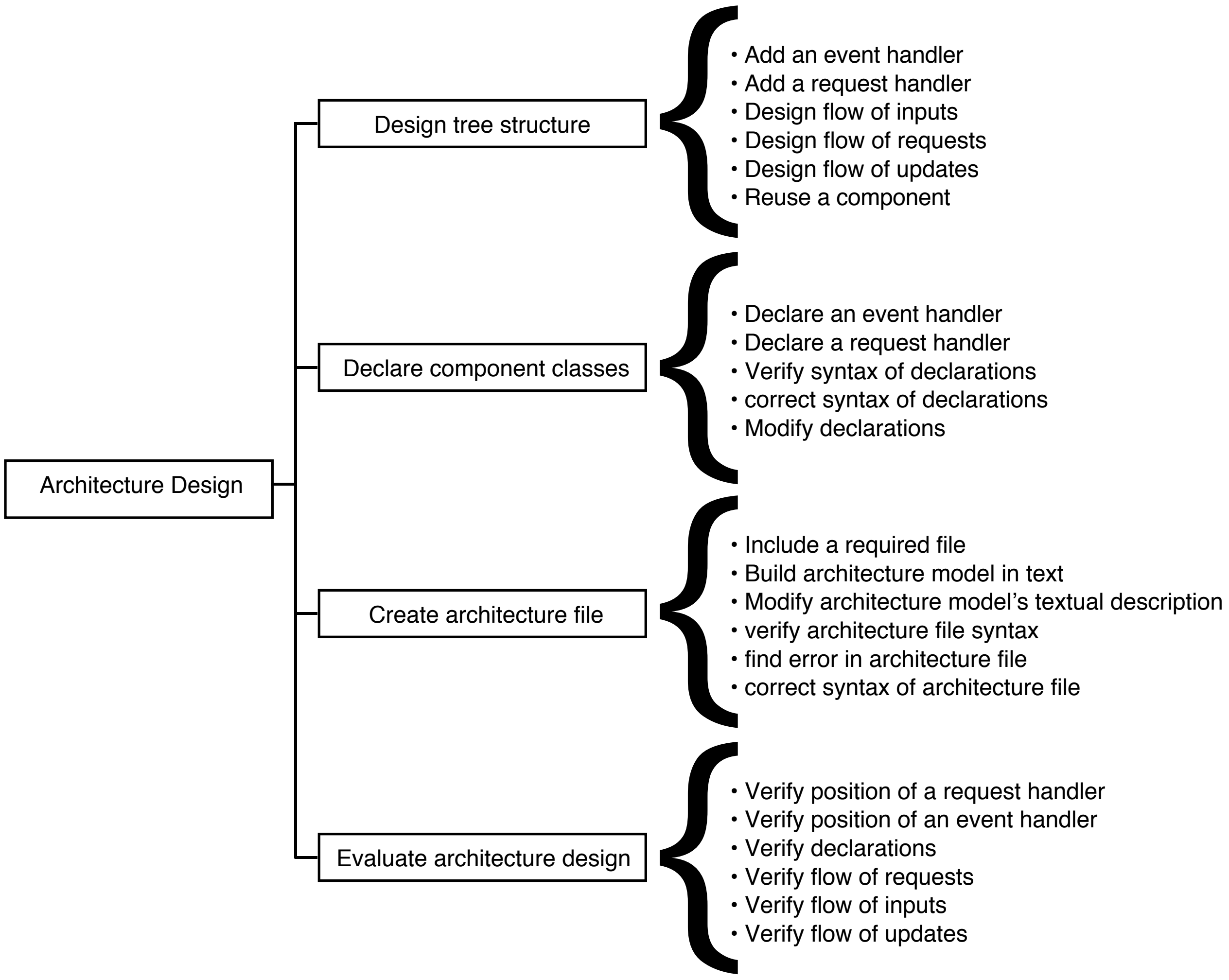

Architecture Design
Design tree structure
• Add an event handler
• Add a request handler
• Design flow of inputs
• Design flow of requests
• Design flow of updates
• Reuse a component
Declare component classes
• Declare an event handler
• Declare a request handler
• Verify syntax of declarations
• correct syntax of declarations
• Modify declarations
Create architecture file
• Include a required file
• Build architecture model in text
• Modify architecture model's textual description
• verify architecture file syntax
• find error in architecture file
• correct syntax of architecture file
Evaluate architecture design
• Verify position of a request handler
• Verify position of an event handler
• Verify declarations
• Verify flow of requests
• Verify flow of inputs
• Verify flow of updates

Figure 4.3 Complete hierarchy of sub-goals and tasks for the Architecture Design sub-goal. The lowest level of this hierarchy lists the individual user tasks associated with each sub-goal.

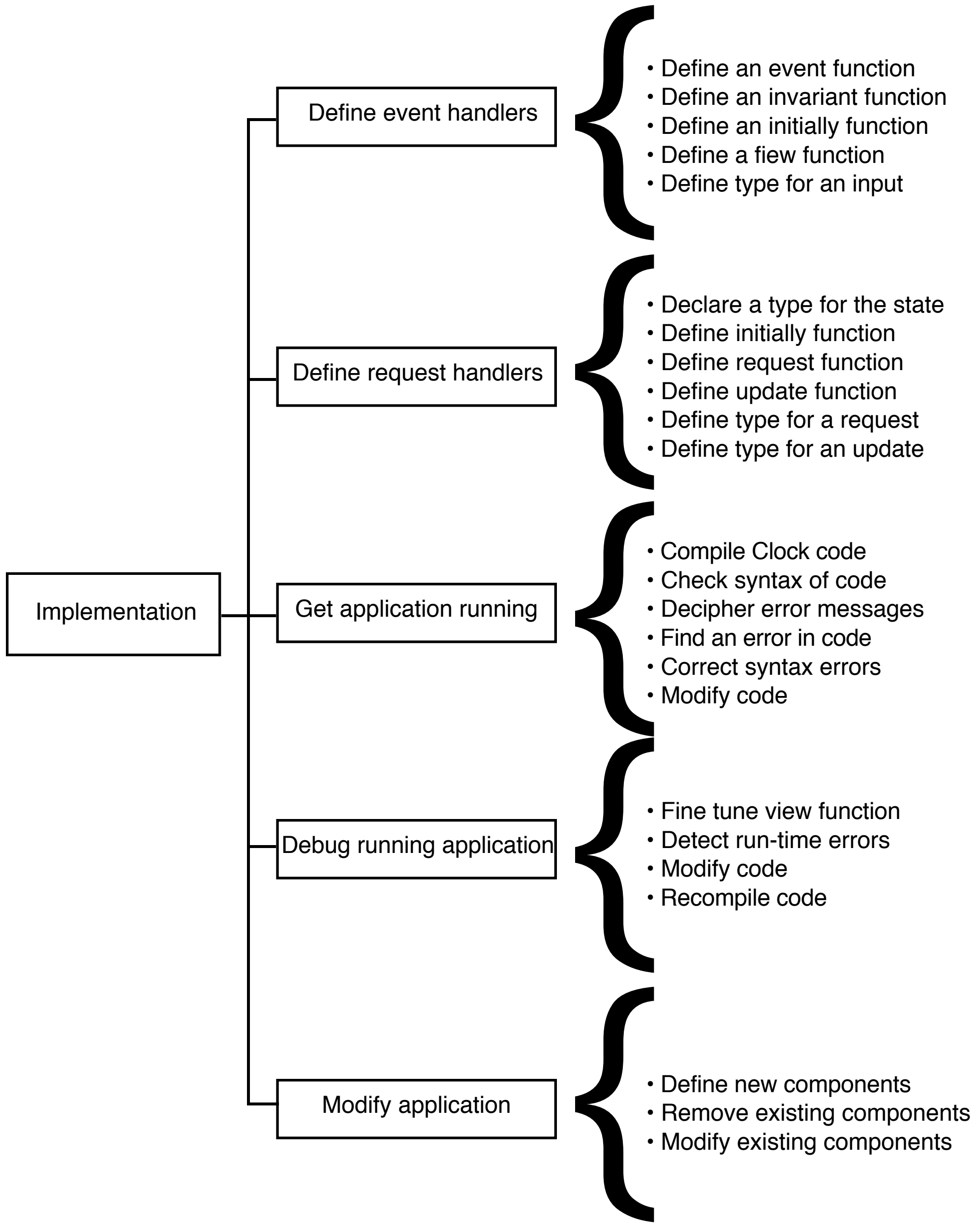

Implementation
Define event handlers
• Define an event function
• Define an invariant function
• Define an initially function
• Define a fiew function
• Define type for an input
Define request handlers
• Declare a type for the state
• Define initially function
• Define request function
• Define update function
• Define type for a request
• Define type for an update
Get application running
• Compile Clock code
• Check syntax of code
• Decipher error messages
• Find an error in code
• Correct syntax errors
• Modify code
Debug running application
• Fine tune view function
• Detect run-time errors
• Modify code
• Recompile code
Modify application
• Define new components
• Remove existing components
• Modify existing components

Figure 4.4 Complete hierarchy of sub-goals and tasks for the Implementation sub-goal. The lowest level of this hierarchy lists the individual user tasks associated with each sub-goal.

The task analysis was refined and finalized after it had been evaluated by a group of Clock users. Since the task analysis was written from the perspective of a single Clock user, it was modified so that it more closely matched the method used by all Clock programmers. There were similarities and differences in the way the various users developed a task analysis:

Everyone used a hand drawing to construct the basic structure of the architecture. The difference was in the amount of detail represented on the hand drawing. Some drew only the basic event handler structure, others added request handlers, and some attempted to add requests, inputs, and updates. All users agreed that the hand drawing became too difficult to follow after a certain amount of complexity was added to it.

The Clock View tool was used for several different purposes by the users. Some didn't use the tool, others used it to verify the syntax of the architecture file, and some used the tool to view the structure and edit the code for each component.

All users reported using incremental development. They would work on the implementation of a specific part of the application and then add other pieces to it. All users wrote very simple view functions for all components which were refined after the application was running.

The case study was an effective way to determine exactly how a Clock program is built using the current system. The task analysis lead to certain conclusions about the current Clock system. Several problems were uncovered which were to be overcome in *ClockWorks*:

Given that the architecture structure is of vital importance to the creation of a Clock program, the current level or support for tasks relating to this structure's design is inadequate.

Currently there can be as many as three different representations of the architecture structure: the hand-drawing, the text file and the picture produced by the CV tool. None of these representations clearly present the information necessary to understand the entire architecture design of a large Clock program. The three representations have to be verified to ensure that they are consistent with each other and with the User Interface design.

The current tools do not verify the paths of requests, updates and inputs in the architecture structure until compile or run time. It is beneficial to have these paths verified before the implementation is begun. Although this verification can be done manually, it is difficult because of the poor quality of the architecture's representation. Programmers often chose not to perform this verification. The result of this is that structural errors in the architecture are found at a stage in development when they are more difficult to change.

The design and implementation of a Clock program is an iterative process. Tasks may be repeated throughout the development of an application. The design of the architecture may be modified at any time during the implementation or run-time verification of the program. It is this style of modification which causes problems throughout the main goal of implementing a Clock program. To use this style of modification, the programmer must be able to understand precisely how the application works. Without a precise and complete representation of how the program functions, it is very difficult to make modifications. The hand-drawn architecture representation quickly becomes too difficult to follow, especially if some time has passed since it was created. The CV tool does not show enough information to explain how the program functions.

The task analysis also identified several problems which were specific to the current implementation of the Clock system. These problems included difficulty understanding error messages, inconvenient limitations with the Clock language, as well as poor performance. These problems are well understood and will be addressed when the implementation of the Clock system and its compiler are modified. These modifications are planned for the Clock system, but they are beyond the scope of this thesis.

4.3 Requirements of *ClockWorks*

Conclusions drawn from the user survey and the task analysis were used to define a set of ten precise requirements for *ClockWorks*.

1. Hide textual details of component class declarations and architecture definition.

The user will no longer need to deal with the architecture file and the component class declarations. Many problems in the original Clock system were caused by problems understanding and debugging these files.

2. Provide full graphical representation of the architecture structure.

ClockWorks will allow Clock architecture structures to be created and modified by direct manipulation. Many of the problems encountered by the users were caused because they could not understand the structure of their program's architecture. This was due to the fact that the architecture had to be completely represented in a textual language before the Clock View tool could illustrate it.

3. Allow direct manipulation of the architecture structure.

To avoid the difficulties of working with the multiple files required to define the architecture structure, users should be able to manipulate the graphical representation of this structure directly.

4. Represent all necessary architecture detail.

Input, request, and update objects will be able to be added and represented in the graphical representation of the architecture. The architecture picture created by the Clock View tool did not display information about inputs, requests, and updates. It was therefore difficult for developers to understand this aspect of their program.

5. Allow Detail to be Abstracted or Hidden

ClockWorks will allow varying levels of information to be displayed for each component. As the architecture structure grows more complex, the amount of information displayed for each component takes up a great deal of screen space. Since users generally work on a single component or group of components at a time, it makes sense that they should be able to hide the information which is not important to their current task. At the same time, they should be able to display all the detail of the component they are working on.

6. Provide a completely integrated programming environment.

The user will be able to compile and run their applications from within the environment. *ClockWorks* will be integrated to support all of the users' tasks without forcing them to exit it for any reason.

7. Provide access to reusable components.

One of the main goals of component-based programming is to allow developers to create components which can be reused in several projects. The Clock system already has library directories to hold reusable components. *ClockWorks* should allow developers to access these libraries and use the pre-defined components in their applications. Also, they should be able to add components to their own libraries.

8. Provide support for architecture verification.

ClockWorks will verify errors in the architecture as it is built. One of the major problems encountered by the users was that the Clock View tool did not verify the architecture. It was possible that many architecture errors were identified at compile or run-time. This forced the users to modify the architecture design at inconvenient times.

9. Allow components to be grouped to form a single group component.

ClockWorks should provide way for users to select a portion of their program's architecture and group it into a single component. This will assist users in managing the detail of their program's structure and also hide detail which is not necessary to understand the group in the context of the architecture structure. This feature will also allow developers to reuse entire architecture sub-trees.

10. Provide simple code generation

ClockWorks should generate generic code for components when they are added so that the architecture will be able to be executed as a prototype version of the application being developed. This feature will allow developers to execute and test individual components. It will ensure that the developers never overlook a function which needs to be defined for a component. Since the architecture will always be in an executable state, Clock developers will easily be able to verify that the structure they have chosen will work. They will also be able to conduct user evaluations of the applications by using the prototype created.

4.4 Conclusions

ClockWorks does not actually need to support all of the tasks identified in the task analysis. This is a consequence of adding support for graphical manipulation of the architecture structure. Several of the tasks identified are eliminated by *ClockWorks*, and some new tasks are added. Tasks relating to the creation of the component class declarations and the architecture file as well as tasks relating to architecture verification were eliminated. Graphical editing tasks, such as cut, copy, and paste were added.

Problems with the current version of the Clock system can be divided into two categories. In the first category are problems associated with understanding Clock architectures. These problems will be overcome by *ClockWorks*. The second category includes problems associated with the current implementation of the Clock system. This category includes problems with incomprehensible error messages, language limitations, and poor performance. In the future, a new implementation of the Clock system is planned. This will include a new compiler and/or interpreter which will eliminate many of these problems. *ClockWorks* will not change the actual implementation of the Clock system. Instead, it will be a tool which is added on top of the Clock system.

The following chapters describe the design and implementation of *ClockWorks*. The ten requirements defined here were used to create the user interface design of *ClockWorks*. They were also used as goals for the evaluation and user testing of the first version of *ClockWorks*.

Chapter 5: The User Interface Design of *ClockWorks*

The task analysis of the Clock programmer helped draw conclusions about the task of developing an interactive program with *ClockWorks*. It was decided that problems associated with the current prototype implementation of the Clock system would not be addressed in *ClockWorks*. The main focus of *ClockWorks* is to overcome problems associated with the creation, manipulation and overall comprehension of the architecture structure of Clock programs.

Component based programming systems require some form of tool support which allows developers to manage the complexity of their program architecture structures. Ten more specific requirements were identified for the Clock system in chapter four. This chapter will illustrate how the user interface was designed specifically to support these requirements.

Section 5.1 explains the prototype instance model which is an important concept used in *ClockWorks*. Section 5.2 explains how *ClockWorks* was designed to meet each of the ten requirements identified in Chapter four.

5.1 The Prototype Instance Model

ClockWorks is based upon the concept of a prototype-instance model [Myers 1992]. In the Clock system, event handlers and request handlers are declared as classes. Instances of these classes are used to construct a program's architecture. Originally, each class had a declaration which listed the names of its attributes (subviews, inputs, requests, and updates). These declarations have been eliminated in *ClockWorks*.

In *ClockWorks*, a class is declared by placing an event handler in the architecture tree, then adding its attributes to it. The declaration of a class is integrated within the architecture tree. The class is modified by manipulating any instance of it. Every request handler and event handler in the tree is both a class and an instance of a class. Modifications to one instance of a class will be reflected in all instances of that class. With this concept, component classes can be declared and instantiated within a single architecture diagram.

5.2 User Interface Design

In the following explanation, the user interface design of *ClockWorks* is explained in terms of the requirements identified in chapter four.

1. Hide textual details of component class declarations and architecture definition

In the original Clock system, the architecture is written in the Clock architecture language (Figure 5.1). This textual description of the architecture is then combined with a declaration for each component class (Figure 5.2). The Clock View tool reads an application's architecture file and displays the general structure of event handlers and request handlers (Figure 5.3).

```
subview monthView =
eh monthView : MonthView
        subview monthShuffle =
        eh monthShuffle : Shuffle
                rh id : Id
                subview leftArrow =
                eh leftArrow : Button
                        rh id : Id
                        rh status : Status
                end eh
                subview rightArrow =
                eh rightArrow : Button
                        rh id : Id
                        rh status : Status
                end eh
        end eh
        subview yearShuffle =
        eh yearShuffle : Shuffle
                rh id : Id
                subview leftArrow =
                eh leftArrow : Button
                        rh id : Id
                        rh status : Status
                end eh
                subview rightArrow =
                eh rightArrow : Button
                        rh id : Id
                        rh status : Status
                end eh
        end eh
        subview weeks =
        eh weeks : Weeks
                rh id : Id
                subview days =
                eh days : Days
                        rh id : Id
                        rh status : Status
                end eh
                subview weekSelectors =
                eh weekSelectors : Button
                        rh id : Id
                        rh status : Status
                end eh
        end eh
        subview monthBar =
        eh monthBar : MonthBar
                subview month =
                eh month : Button
                        rh id : Id
```

```
                              rh status : Status
                    end eh
          end eh
end eh
```

Figure 5.1 The Clock Architecture Language.

```
% Button Event Handler Class
ehClass Button
          inputs mouseButton
          requests myId, isSelected
          updates setMyId, select, unSelect, doAction

% Id Request Handler Class
rhClass Id
   requests myId
   updates setMyId
```

Figure 5.2 Component Class Declarations.

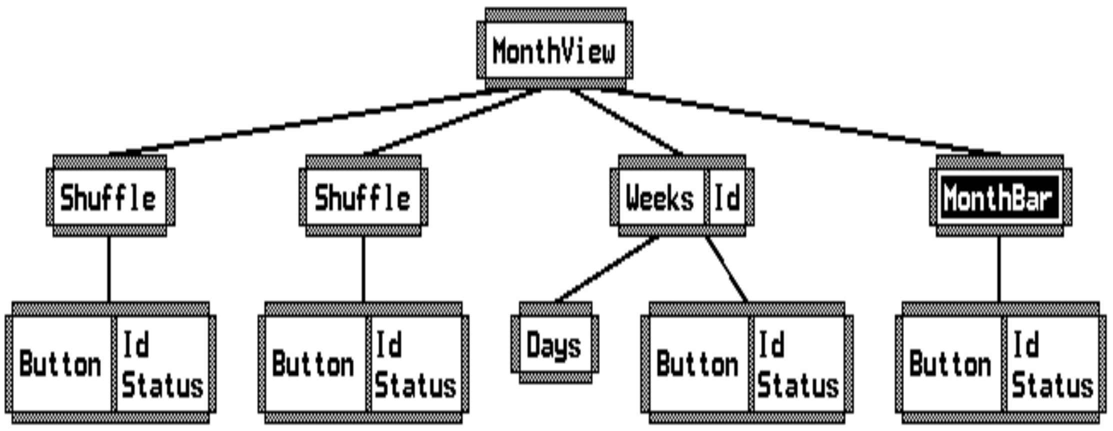


Figure 5.3 Architecture Tree Drawn by Clock View Tool.

ClockWorks hides the detail of the component class declarations and the textual representation of the architecture from the user. Instead, *ClockWorks* opens an empty architecture window upon which the developer can build an architectural model for the project being developed. The architecture is represented and manipulated through a picture.

2. Provide full graphical representation of the architecture structure

A node in the architecture tree consists of an event handler component, any number of request handler components, and the inputs, requests, and updates sent and received by the components making up the node. Event handler and request handler components are drawn as boxes on the screen. Event handlers have two labels. The first label is the name of the subview being filled by that component and the second label is the name of the event handler class associated with the component. All event handlers (except the root of the tree) have a line connecting them to their parent in the tree. Request handlers are boxes attached to the bottom of event handlers. Each request handler has one label displaying its class name. Figure 5.4 illustrates how components are represented in *ClockWorks*.

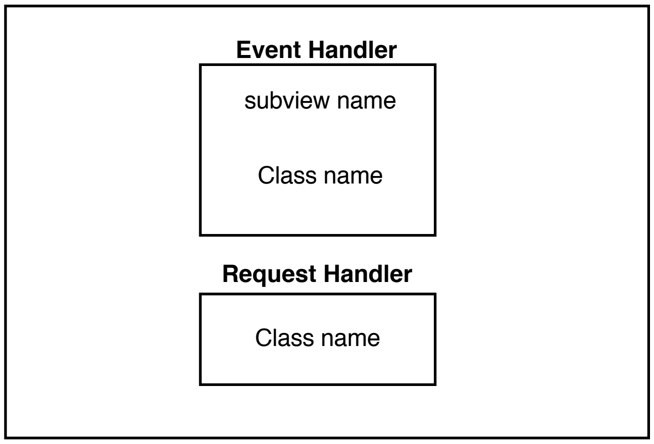


Figure 5.4 Components in ClockWorks .

The term interface is used to refer to the inputs, requests, and updates (iru events) received and sent by a component. These are attached to the left or right side of the box which represents the component with which they are associated. Iru events received by a component are drawn on the left side, while those sent by the component are drawn on the right. Figure 5.5 illustrates how iru objects are drawn, and Figure 5.6 shows how the *Shuffle* component (explained in Chapter 2) is represented in *ClockWorks*.

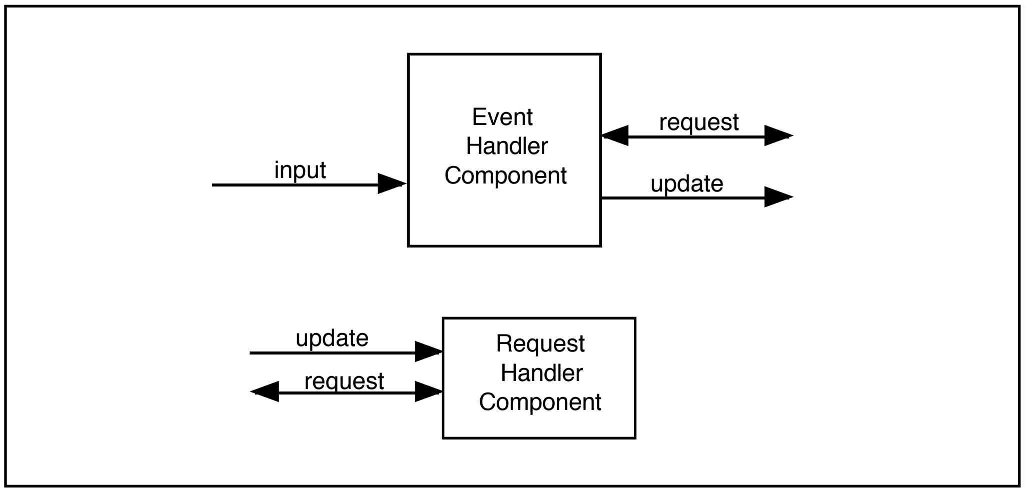


Figure 5.5 Interface Events (inputs, requests and updates) in ClockWorks. *Requests are always represented as an arrow with two heads and a label which is the name of the request. Since updates and inputs are essentially equivalent in Clock, they are both represented by an arrow with one head and a label which is the name.*

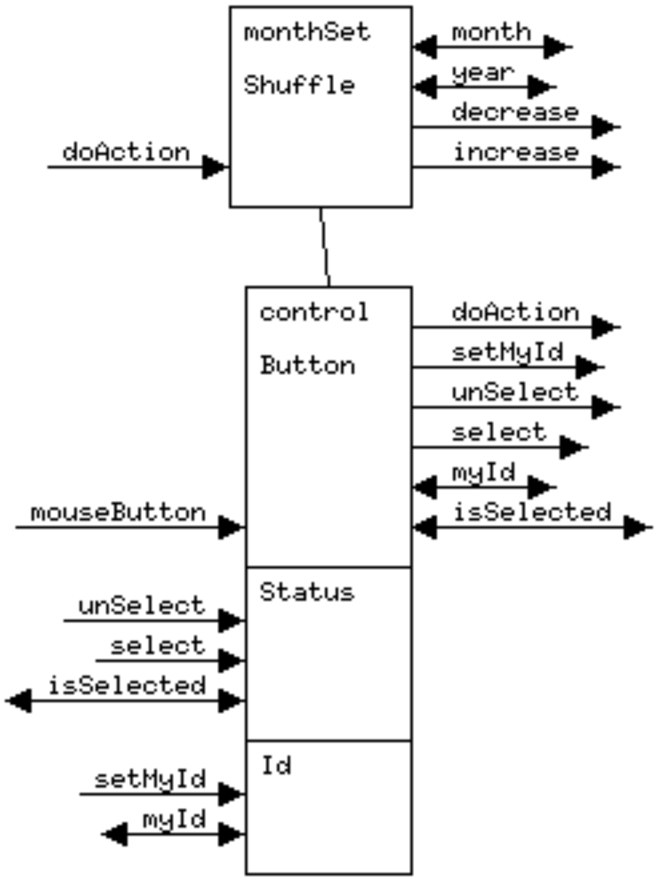


Figure 5.6 The Shuffle Component. This component is explained in chapter 2.

3. Allow direct manipulation of the architecture structure

To avoid the difficulties of working with the multiple files required to define the architecture structure, users should be able to manipulate the graphical representation of this structure directly.

All of the commands used by the developer are listed in menus at the top of the architecture window. There are eight menus (File, Edit, Detail, Class, Component, Group, Library, Test). Each one contains a different category of commands. Figure 5.7 illustrates all of the commands available in the menus. Commands listed in italics have not been implemented in the first version of *ClockWorks*. The class and instance menus will be explained in this section while the other menus will be explained in the section devoted to the requirement they support.

File	**Edit**	**Detail**	**Class**	**Instance**	**Group**	**Library**	**Test**
Open Project	Remove	Open	Rename	Add request handler	*Group / Ungroup*	Add to Library	Compile
Close Project	*Cut*	Close	Duplicate	Replace			Run
Save Project	*Copy*	Interface/ hide interface	Add subview				
Quit	*Paste*	From	Add Root				
		Root	Add request				
			Add update				
			Add input				
			IRU Declaration				

Figure 5.7 Menu Commands of ClockWorks.

The *class* menu contains all commands which will affect the class declaration of the currently selected component. The actual class declaration is hidden from the user of *ClockWorks*. Classes are declared by manipulating an instance of the class.

The *rename* command is used to change the name of a class. It works slightly differently for each type of object. In each case, a dialog box appears asking for a new name.

There are two possible name changes which can be made for an event handler component. Changing the class name will give every instance of this class a new name. Changing the subview name will give every instance of this subview a new name. The subview name is part of the class declaration of the parent of the currently selected event handler. Therefore, changing the subview name will modify every instance of the parent's class.

Renaming a request handler will give every instance of this class a new name. Renaming an input, request or update object will give every instance of this object a new name.

The *duplicate* command is used to create a new class with the same declaration as the currently selected component. The class may then be modified without affecting the original class. The *duplicate* command can also be used to create a new input, request, or update object which has the same definition as the original one.

The *add subview* command adds a new event handler as a subview of the currently selected event handler. If there is no architecture tree defined, the add subview command creates the root of the tree. When a new event handler is added, the user will be required to enter a subview name and a class name for the event handler. The user may select a previously defined class or create a new class by entering a new name.

The *add root* command will create a new event handler as the parent of the root of the architecture tree. It will add the first node in the tree, if no tree has been previously defined.

The *add request* and *add update* commands are used to add a request or an update to the currently selected event handler or request handler. Note that requests and updates are sent by event handlers and received by request handlers. A special update which is received by an event handler is called an input. The *add input* command allows an input to be added to the currently selected event handler.

When adding inputs, requests, or updates, the user will be required to enter a name for the object. A user may select a previously defined name or enter a new name. Inputs and updates also require a list of types for the arguments taken by the object. If the object takes no arguments, this field may be left blank or the type Void may be entered. Each request requires a type for the value it returns and a list of types for the arguments it takes.

The *IRU Declaration* command brings up a dialog box which allows the user to modify the type declarations for an iru object. Note that all iru objects with the same name must have the same types defined. Therefore, the *IRU Declaration* command will affect all instances of the selected input, request or update (iru) object.

The *Component* menu contains all the commands which affect the definition of an instance of a component in the architecture.

The *add request handler* command allows the user to add a request handler to the currently selected event handler. The user will be prompted to enter the class name for the request handler. This name may be selected from a list of request handler classes which are already defined, or the user may enter a new name to create a new request handler class.

The *replace* command allows you to replace the currently selected object with another one of the same type. An event handler can only be replaced with another event handler which has the same subviews. Request handlers can be replaced with any request handler class. The replace command has no effect on iru objects.

In the first version of *ClockWorks*, the *edit* menu contains only the *remove* command. The user can choose to remove (or delete) the currently selected object. If an event handler is removed, the entire subtree rooted at that event handler is removed from the tree. If a request handler, update, input, or request is removed, there is no effect on the rest of the tree.

In future versions of *ClockWorks*, users will be able to select a connected group of components from the architecture tree. The *edit* menu will then contain the traditional *cut, copy, and paste* commands. The addition of these commands will give the users more freedom to work with the architecture tree. The *cut* command will allow a group of components to be removed from the tree at once. The *copy* command will allow users to select and copy a section of the tree. The *paste* command will allow the users to attach a section of tree in another position.

4. Represent all Necessary Architecture Detail

ClockWorks has defined a way to graphically represent every kind of element which can be part of a program's architecture. There are five distinct types of elements which are: event handlers, request handlers, inputs, requests, and updates.

With all of these details displayed, the users will be able to see the structure of the components making up their programs as well as the way in which these components communicate.

5. Allow Detail to be Abstracted or Hidden

Once an architecture structure gets to be a significant size, the amount of detail associated with it is too difficult to follow. The user of *ClockWorks* can choose the amount of detail displayed for each component in the architecture structure. With this feature, the developer will not be overwhelmed with the amount of detail associated with a large Clock architecture. The commands which are part of the *detail* menu allow the user to hide or abstract information for components.

The *detail* menu contains all the commands which affect the amount of detail represented for architecture components.

The *open* command is the same as a double click on a component. It increases the level of detail for a component. The *close* command decreases the level of detail. Figure 5.8 illustrates the various levels of detail for each type of object in *ClockWorks*. Figure 5.9 shows the appearance of an event handler component at various detail levels.

	Event Handler	***Request Handler***	***Group***
OPEN t			
	HIDDEN		*HIDDEN*
default	*NORMAL*	*NORMAL*	*NORMAL*
	RH	*EDIT*	*RH*
	EDIT		*VIEW*
CLOSE s			

Figure 5.8 Detail Levels for Clock components. This table illustrates how the detail level for each type of component is changed with the open and close commands. The open *command increases the level of detail to the next level (down the table), while the* close *command decreases the level of detail (up the table). Since the* open *and* close *commands do not apply to iru objects, they are not included in this table. Note that group objects are not included in the first version of* ClockWorks. *The default for each type of object is the NORMAL state.*

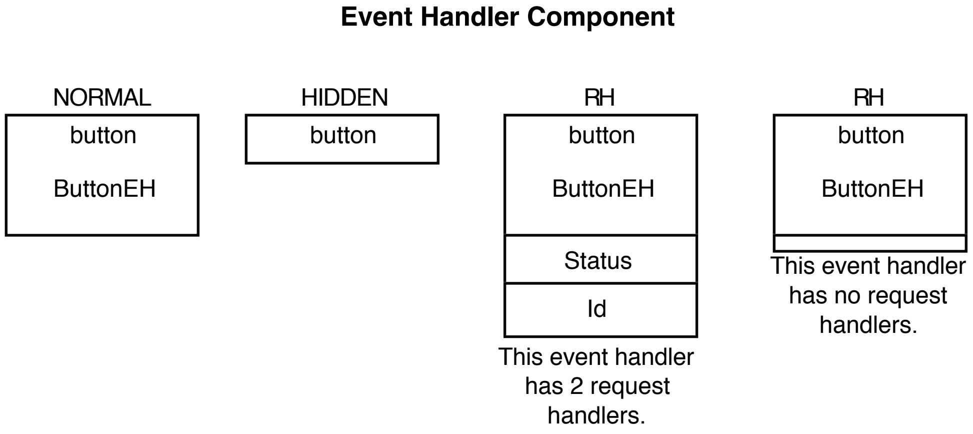


Figure 5.9 Event handlers at various detail levels. The HIDDEN *state indicates that the sub-tree rooted at this object is hidden. The object in the* HIDDEN *state will have only a single label which is the name of the subview instantiated in that component. The* RH *state indicates that the request handlers attached to the given object are displayed. If there are no request handlers attached to the object, a small extension is added to the bottom to indicate to the user what state the object is in. The* EDIT *state indicates that the editor is open for the object.* ClockWorks *uses the editor indicated in the* EDITOR *environment variable. The default editor is* XEdit.

The term interface is used to refer to the various inputs, requests, and updates attached to an event handler or request handler. This information can be turned on or off by selecting the *interface/hide interface* menu command. This command works as a toggle. If the interface is already on, this command turns it off. If the interface is off, this commands turns it on.

The *From* command operates on the currently selected event handler component. It makes this event handler the root of the displayed architecture. Only the sub-tree rooted at this event handler will be displayed. The *Root* command is the opposite of the *From* command. It resets the tree to its original root.

6. Provide a completely integrated programming environment

In the original Clock system, the developer worked from the command line in a UNIX environment calling each tool separately and maintaining a list of text files which defined the application being developed. One goal of the *ClockWorks* was to create a fully integrated tool which will support all phases of the architecture design and implementation of a program. It is hoped that the general design of *ClockWorks* will help to make the Clock system easier to learn and to use. Commands in the *file* menu allow users to access their architecture file while commands in the *test* menu allow the project to be compiled or run.

The *file* menu contains the commands which allow programs to be opened, closed, and saved. Also, it contains the command to exit *ClockWorks*. The *open project* command prompts the user for the name of a project. If the project already exists, its architecture is opened, otherwise a new project is created with the given name. The *close project* command allows the user to close the current project without quitting *ClockWorks*. The user has the option to save the project before closing or close without saving. If the project does not have a name, the user will have to specify a name before saving. To save the architecture file, the user chooses the *save project* option. If the project has not been given a name, the user will be asked for a name which has not already been used for another project. To exit from *ClockWorks*, the user will choose the *quit* command. The user will be given the option to save before quitting or to quit without saving. If the project does not have a name, the user will have to specify a name before the project can be saved.

The *test* menu contains the commands to compile and run the program being developed. The *compile* command simply issues the command 'updatearch' for the project being developed. The 'updatearch' command was used to compile Clock programs in the original Clock system. In the first version of *ClockWorks*, compiler errors are displayed on the terminal from which *ClockWorks* was run. Once the implementation of the Clock system has been updated, these error messages will be printed in a scrolling message window which will appear directly below the window which displays the architecture drawing.

The *run* command simply issues the Clock command 'go' for the project being developed. The application will then run alongside *ClockWorks*. Again, error messages are displayed on the terminal from which *ClockWorks* was run.

7. Provide access to reusable components

One of the main goals of component-based programming is to allow developers to create components which can be reused in several projects. The Clock system already has library directories to hold reusable components. There is a system library which contains declarations and code for components which can be accessed by all Clock users. In addition, each Clock user has a local library to which components can be added. *ClockWorks* will allow developers to access these libraries and use the pre-defined components in their applications. They will also be able to add components to their own local library of components.

This feature allows any Clock programmer to create standard interface components such as buttons, editfields, and scroll bars store them in a library. With a large enough library of components, it is possible that an entire application could be built from previously-defined components.

8. Provide support for architecture verification

ClockWorks will verify errors in the architecture as it is built. One of the major problems reported by Clock users was the lack of support for architecture verification. Errors in the architecture were often not found until implementation was begun. *ClockWorks* has two features designed to overcome these problems. These features were not implemented as part of the first version of *ClockWorks*.

As each input, request or update is added to the architecture, *ClockWorks* will search the tree for the component which either handles it or makes it. If the source or destination of the iru object cannot be found, it will be drawn with a question mark. With this feedback, the user will be able to immediately tell if an iru object has been declared that is either not used or not handled in the tree.

By clicking on an iru object, *ClockWorks* will search the tree for the component which is its source or destination. If this component is found, it will be highlighted so that the developer will be able to locate it.

9. Allow components to be grouped to form a single group component

With the grouping feature, the developer will be able to select a node or group of nodes from the architecture structure and group them so that they appear as one node in the tree. This has three advantages.

First, once the developer has completed the internal details of a particular section of the architecture, these internal details can be hidden. Once the section is functioning correctly, there is no need to see the requests, updates, and inputs which are sent and received within the group. When the interface of a group component is shown, only requests, updates, and inputs which are sent to or from components outside the group are displayed.

Second, the developer may define a particular program component and group it into a single architecture node once it is complete. Any node in the tree can be instantiated with this group. For example, the developer may define a section of the architecture which represents a scroll-bar. Once it is functioning, the scroll-bar architecture can be grouped into a single component which functions as a scroll-bar. If the developer needs another scroll-bar in some other section of the architecture, it can easily be added by instantiating a single subview with the group component. Without this feature, the scroll-bar would have to be reconstructed one component at a time.

The third advantage is that developers will be able to reuse entire sub-trees of an architecture rather than only a single component. A group component can be selected and added to a library.

The addition of the grouping feature requires some minor modifications to the current implementation of Clock so that it will recognize a group as a new kind of component. A group component will look similar to an event handler and its definition will be similar. Users will be able to attach subviews, request handlers, inputs, requests, and updates to group components. The difference between a group and an event handler is that the internal details of a group include an actual architecture structure. Figure 5.10 shows how the shuffle component defined in figure 5.7 would appear as a group. Once a group is created, it can be modified the same way event handlers are modified.

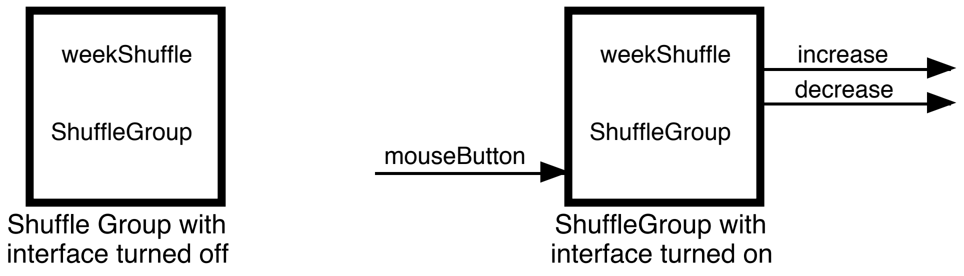

weekShuffle
ShuffleGroup
Shuffle Group with interface turned off
weekShuffle
ShuffleGroup
increase
decrease
mouseButton
ShuffleGroup with interface turned on

Figure 5.10 The Shuffle group component. When the interface of a group is turned on, only those inputs, requests, and updates which are sent or received from outside the group are displayed. This gives enough information to understand how to place the group in the context of the architecture structure. Group components will be drawn with a thick border to distinguish them from event handlers.

When a group is opened (with a double click or by selecting *open* from the *detail* menu), the internal details of the group will be displayed. The user will not be able to modify the definition of the group unless it is ungrouped.

10. Provide simple code generation

Each component class is defined in Clock as a series of functions which are called in response to events. An advanced feature of *ClockWorks* was designed to assist in the creation of executable prototype versions of applications. As a new component class is being created and modified, *ClockWorks* will generate default code for the functions which define it. This feature will assist the developer in three ways.

First, the architecture of the application will always be in an executable state. The developer can therefore test the application before it has been completely implemented. Modifications to the architecture can be made before they become costly.

Second, the developer will be able to fill in the details of a single component or group of components in the tree without the added complexity of having to define all the components before a single one can be tested. This will allow the developer to work on the definition of one component while not having to worry about how other components are defined. In this way, an application can be implemented incrementally.

Third, *ClockWorks* will assist the developer in creating a correct architecture because it will fill in all the required functions for each component based on how the architecture of that component has been defined. The developer will be able to see each function which must be defined in order for the component to work properly. It is less likely that the developer will get deeply into the implementation and compilation stages of application development and still discover details which were overlooked in the architecture design.

It is hoped that this support for code generation will assist the developer in the task of developing interactive applications with the Clock system. It simply fills in default values which must be included in the definition of each component class in order for it to run. *ClockWorks* will attempt to automate as much of the development process as possible.

5.3 Conclusion

The design presented in this chapter is the final design after evaluation and modifications were complete. The design went through many different forms throughout the entire development cycle (including implementation and user testing). The results of these evaluations are discussed in chapter 7. Appendix B illustrates the UAN description of the tasks supported by *ClockWorks*. Most major modifications were made very early in design when it was still on paper and very easy to modify. Modifications made during implementation involved smaller details such as the order or name of menu commands.

The user interface design is the conclusion of a number of development steps which involved evaluation of the existing system and analysis of the user tasks to be supported. From this point, the development of *ClockWorks* moves its focus from design to implementation. Unfortunately, the existing version of Clock does not support enough features to allow *ClockWorks* to be developed in Clock. Therefore, the implementation stages will not as closely follow the steps of the Clock methodology.

Chapter 6: Implementation

The evaluations which took place throughout the design of *ClockWorks* all lead to the conclusion that it would fulfill the requirements identified for it. However, without actually using the system, these points cannot be proven. In order to be able to conduct more formal user testing, *ClockWorks* was implemented and tested with its users. With a working version of *ClockWorks*, formal user testing was performed in order to prove that the design would adequately support the tasks required by the users and make component-based programming an easier and more effective way to build interactive applications.

This chapter will describe several important aspects of the implementation of the first version of *ClockWorks*. Since *ClockWorks* was designed to be easy to learn and use, the implementation had to deal with many of the complexities which are being hidden from the users.

ClockWorks was implemented in C using Motif widgets to build its interface. Clock is not yet powerful enough to be used to create its own programming environment.

6.1 Interface with Clock

ClockWorks was designed to interface with Clock by creating all the information it needs to compile and run an application. In the original Clock system, users produced an Architecture file written in the Clock Architecture language (see figure 5.1), as well as the declarations and functional code for each component class used. The Architecture file and the component class declarations are combined and translated to form a new architecture file which is written as a list of constraints. The compiler takes the Architecture constraints and the component class definitions and creates the executable file *Arch.x*. The architecture file uses the architecture constraints to run the program. Figure 6.1 illustrates this structure.

ClockWorks simplifies this structure. It creates the Architecture constraints file directly from user interactions with a graphical architecture structure. Figure 6.2 illustrates how *ClockWorks* fits into the Clock System.

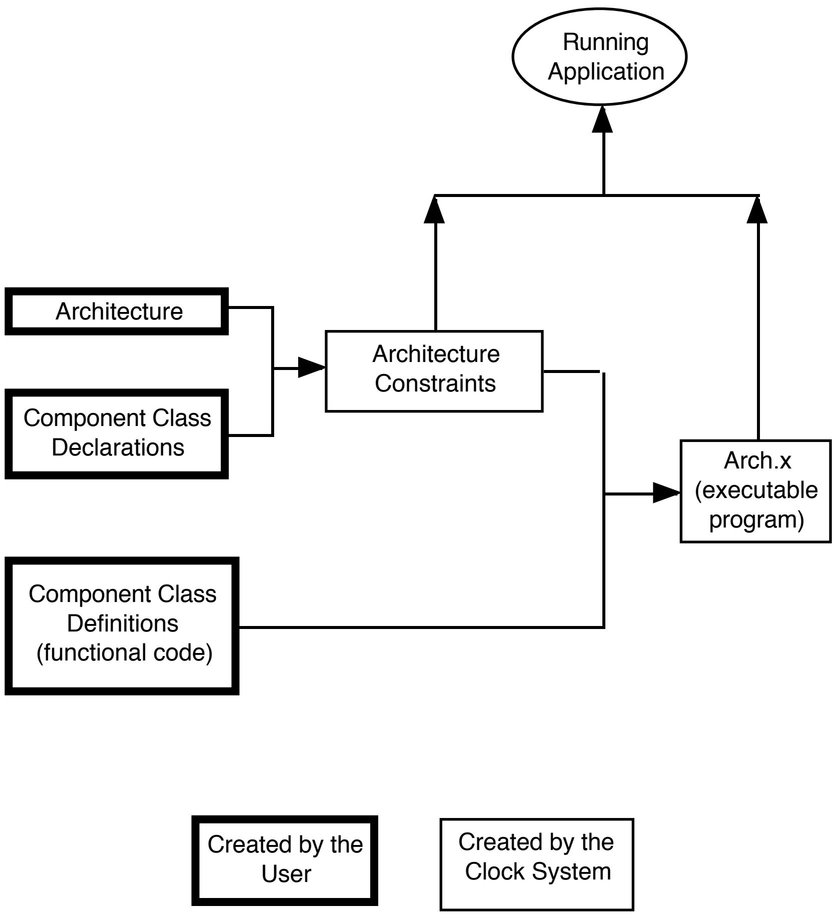
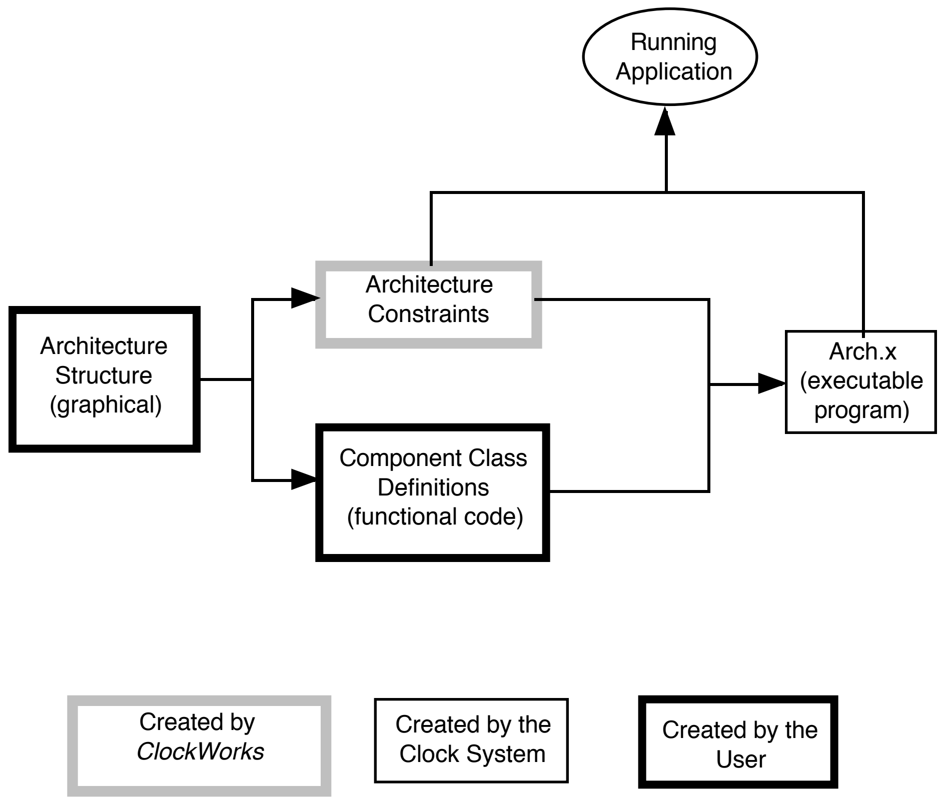

Running Application
Architecture
Component Class Declarations
Architecture Constraints
Arch.x (executable program)
Component Class Definitions (functional code)
Created by the User
Created by the Clock System

Figure 6.1 Structure of the Original Clock System.


Running Application
Architecture Structure (graphical)
Architecture Constraints
Component Class Definitions (functional code)
Arch.x (executable program)
Created by *ClockWorks*
Created by the Clock System
Created by the User

The Clock Programming Environment as part of the Clock system

Figure 6.2 ClockWorks *as part of the Clock System.*

The Clock system can have new development tools added to it very easily as long as the architecture constraints file and the component class definitions are created. Since *ClockWorks* does not interface with the compiler directly, it will not be affected by changes or updates to the Clock compiler which are planned in the near future.

When *ClockWorks* needs to create a new project, compile a project, or run a project, it simply issues the same commands the user originally used for these tasks. When a new project is given a name, *ClockWorks* runs the *mkarch* command which creates the directory for the project in the developer's Clock directory. When the *compile* command is chosen from the *test* menu, *ClockWorks* saves the architecture file and calls the *updatearch* command. When the *run* command is chosen from the *test* menu in *ClockWorks*, the *go* command is executed.

ClockWorks creates the files which are used by the system to compile and execute the programs. It does not directly interact with the compiler and will therefore not be affected by updates or modifications to the compiler.

The user is no longer required to produce detailed textual descriptions of component classes and the architecture structure. The user interacts only with a graphical representation of the program's structure. In the future, new tools will be created to assist the developer in creating the functional code which defines the component classes. These new tools can be added to the Clock system and work with *ClockWorks*. Chapter 8 discusses some of the future plans for the Clock system.

6.2 Screen Layout

The main screen of *ClockWorks* is a compound Motif widget which controls a menu-bar, a work area, and a message window. As the basis of *ClockWorks*, it was the first thing implemented. Figure 6.3 shows the main window of *ClockWorks* with the architecture of the interactive calendar program displayed in the work area.

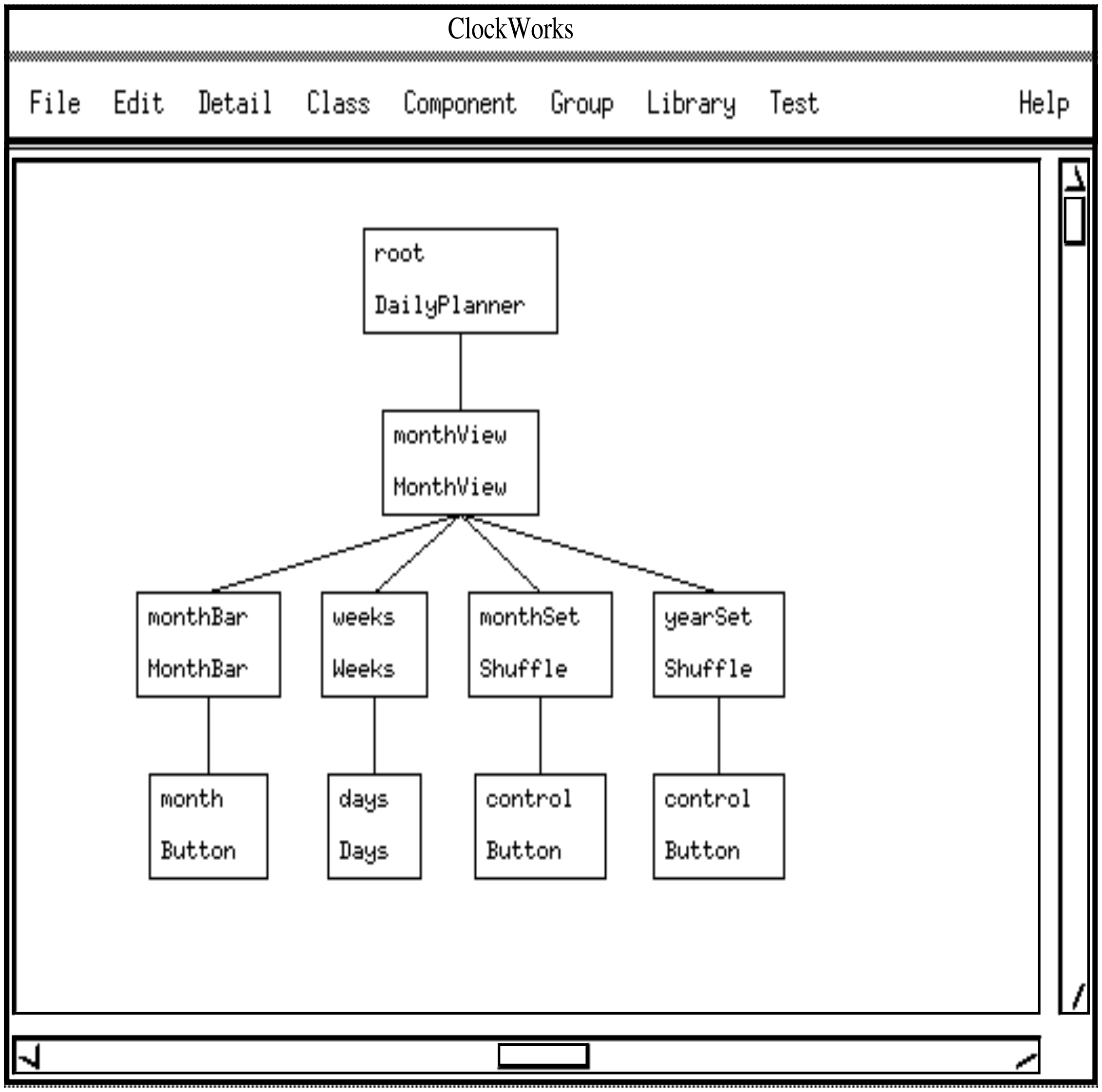


Figure 6.3 The Main Window of ClockWorks *with the Architecture of the Interactive Calendar displayed.*

All the commands available to the user are contained in a menu bar on the top of the main screen.

The architecture is drawn on a large canvas. A portion of this canvas is visible in the scrolling window which makes up the work area of the main screen. *ClockWorks* is able to display more information than the Clock View tool. When shown in full detail, a node takes up a large part of the canvas. Nodes are drawn as small as possible in order to allow as much of the architecture as possible to fit into the scrolling window.

The message window is provided for a future implementation of the Clock system and its compiler. It will be used to display messages to the user concerning the compilation and running of the program. However, the message window is not currently used or displayed in *ClockWorks*. Messages from the compiler are displayed on the user's X terminal.

All of the user's tasks may be performed from the main window. However, some commands bring up a second window. Any command which requires the user to enter textual information for a component such as its name, class, or type signature, causes a dialog box to be displayed. The information is entered into text fields in the dialog box and the user presses a button to complete the command. When the user pushes the OK button all of the data entered is verified. The information is not connected to a component until the OK button has been pushed and the dialog box is dismissed. The dialog box is not dismissed unless all the information entered is valid. An attempt has been made to provide error messages which indicate the error which has been made.

There are only three different dialog boxes used in *ClockWorks*. There is one for each type of architecture element. These types are event handler, request handler, and iru events (input, request and update). Each dialog box allows the user to enter all the information associated with the element being declared. Several commands require only part of this information. When these commands are used, the dialog box is modified so that it only allows the appropriate information to be entered. Figures 6.4, 6.5 and 6.6 illustrate the dialog boxes used for event handlers, request handlers, and iru events respectively. These figures show the full dialog boxes which allow all the information to be entered. Each dialog has a list of the previously defined component classes or iru objects. These are the names of the objects already defined for the current architecture as well as the names of objects defined in both the system library and the user's own local library of reusable components.

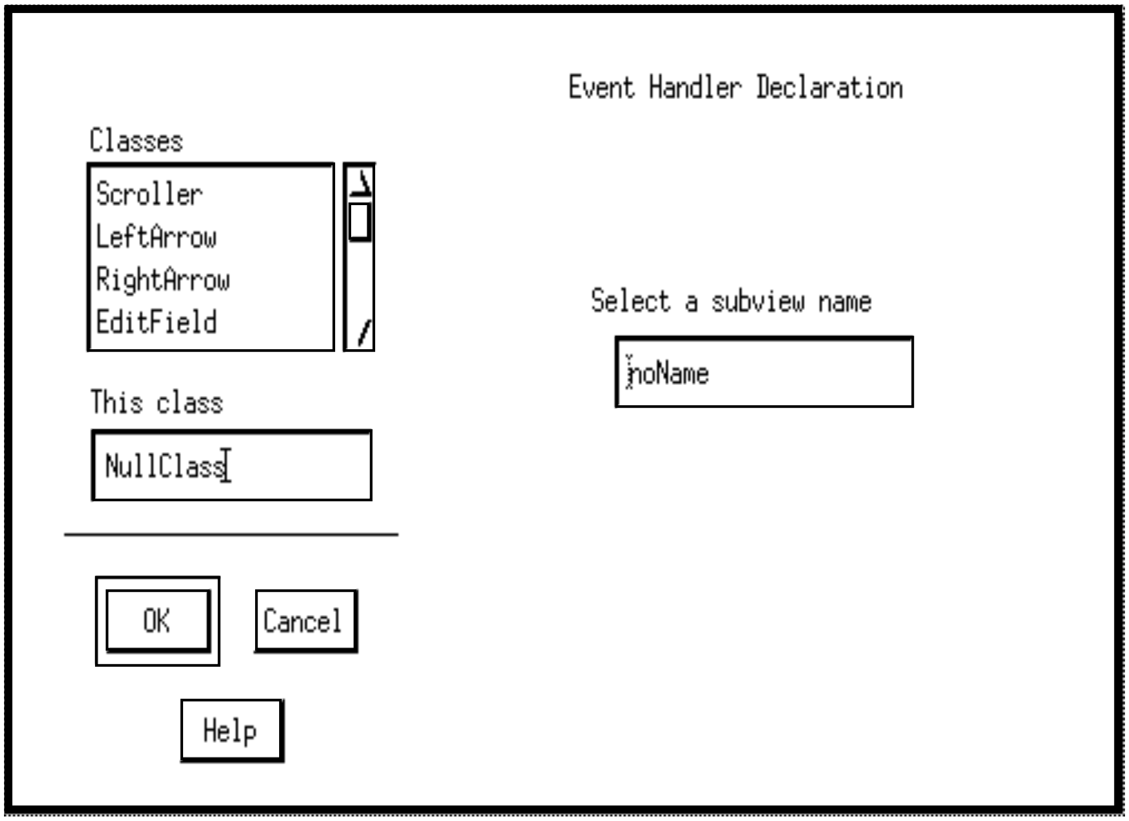


Figure 6.4 Event Handler Declaration Dialog Box. This dialog box allows the user to enter a class name and a subview name. The names of previously defined classes appear in the listbox. Developers may use any of the pre-defined names to instantiate an event handler, or they may choose a new name to define a new request handler class.

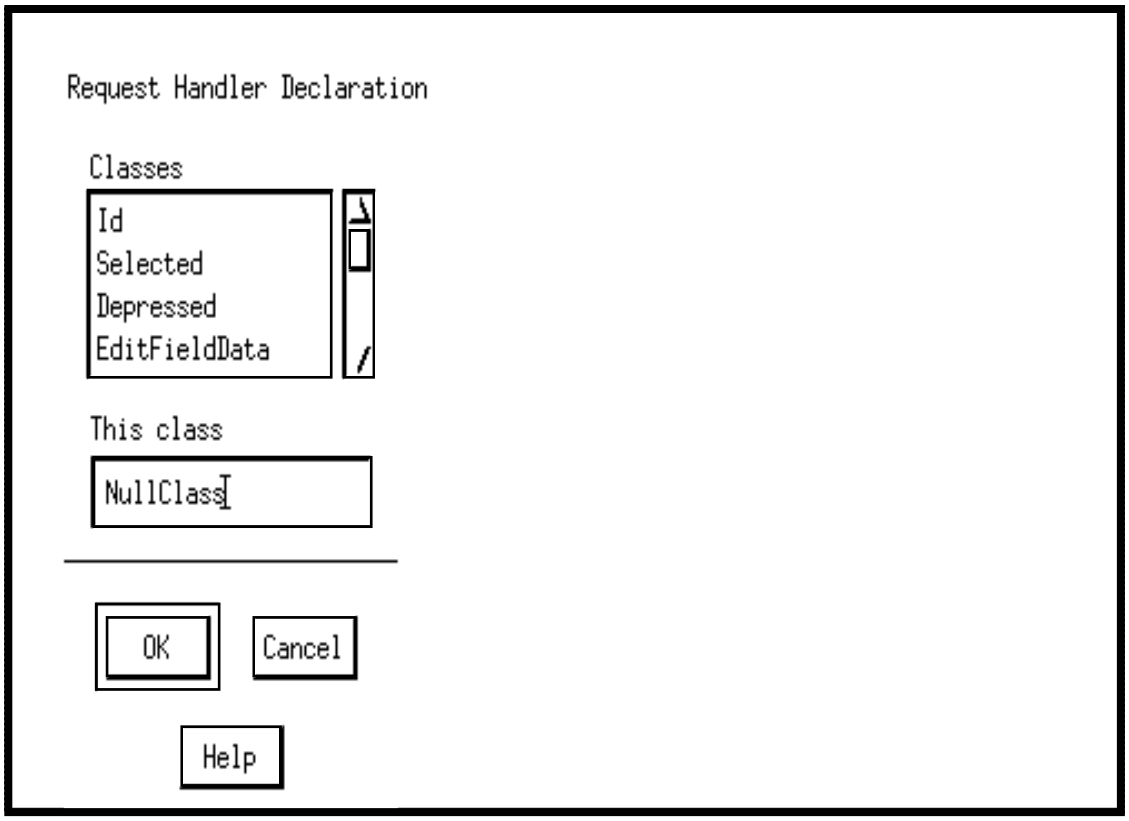


Figure 6.5 Request Handler Declaration Dialog Box. This dialog allows the user to enter a class name for the request handler. The names of previously defined request handler classes appear in the listbox. Developers may use any of the pre-defined request handlers classes to instantiate a request handler, or they may enter a new name to define a new request handler class.

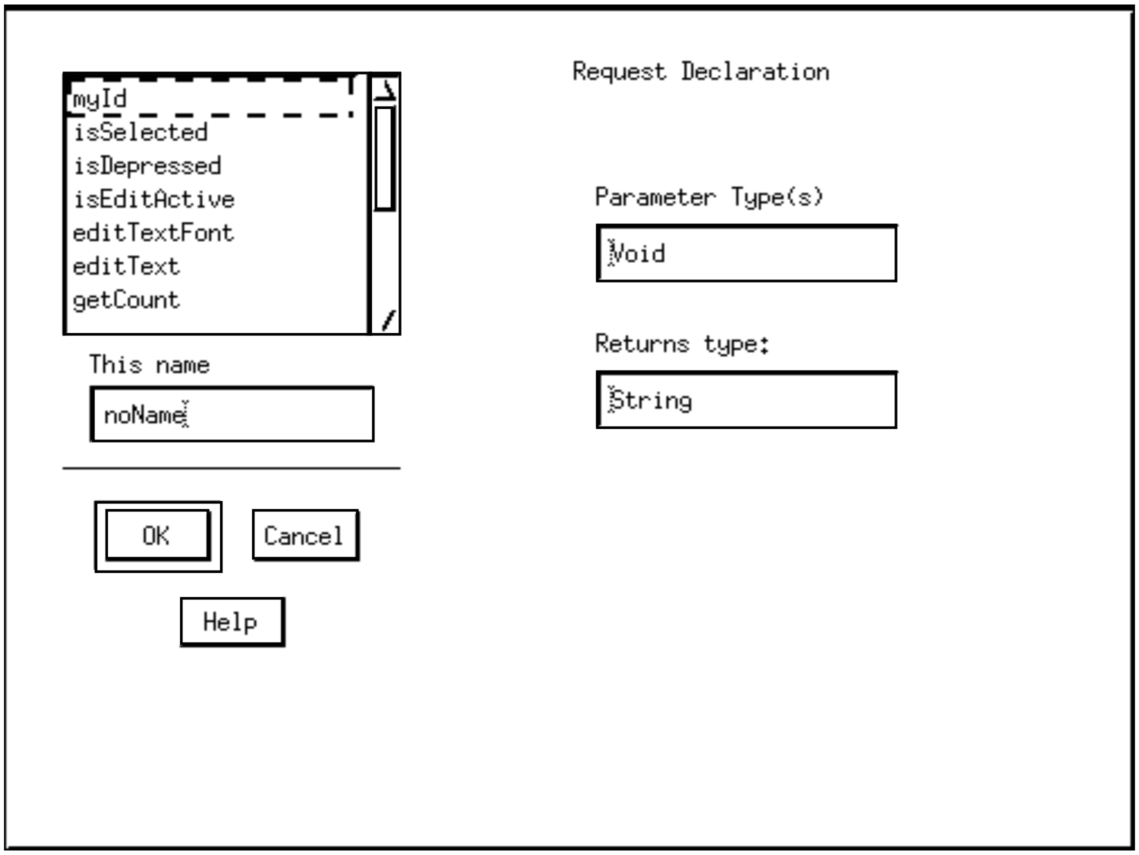


Figure 6.6 Input, Request and Update Declaration Dialog Box. This box is specifically for request objects. The dialog box for update and input objects does not include a field for the returns type. Each iru event needs a name, and a list of parameter types. Request events also require the type to be returned. The names of previously defined iru events appear in the list box. Users may choose any of these names, or they may choose a new name to define a new iru event.

An editor is opened whenever a user decides to open a component for editing. *ClockWorks* chooses the editor which is in the environment variable EDITOR. The editor has its own window and runs as a separate process so that the architecture may be modified while an editor is open, and so that more than one component can be edited at a time.

Whenever a Clock program is run, its window is opened as a separate process. The architecture can be manipulated while the application is running, although changes will not affect the running application. It must be compiled and run again.

6.3 Incremental Implementation

ClockWorks was almost always in an executable state throughout its implementation. This was done to facilitate user testing and to make the process of implementation easier. The implementation process went through four distinct phases before the first version was complete.

The first phase of the implementation was to provide a tool with which a picture of an architecture tree could be built. Although this first phase did not assign any meaning to the picture, it gave the users an idea of how the interface could be used to build Clock architectures.

In the second phase of implementation, the functionality to open existing architecture constraint files was added. In the original Clock system, these constraint files were built by the CV tool. Once *ClockWorks* could open these files, the users were able to open the architecture structures of their existing Clock projects. They could then evaluate *ClockWorks'* method of displaying architecture trees.

In the third phase of implementation, commands were added so that the users could save architecture structures from *ClockWorks*. Once this functionality was added, Clock programs could be compiled and run from within *ClockWorks*.

The fourth phase of implementation involved adding the prototype instance model and the internal details through which component classes were declared. Once this phase was complete, the users were able to build new architectures, or modify existing ones. From this point on, the users were able to use *ClockWorks* to create Clock applications.

The remaining sections of this chapter will deal with some of the internal details of the implementation of *ClockWorks*.

6.4 Architecture Structure

For the sake of clarity, the term *object* will be used to refer to the elements which are displayed in the architecture drawing. These objects are event handlers, request handlers, and iru events (input, request, update).

A single data structure called *tObject* is defined to hold all the information for one architecture object. All objects are stored in the same data structure. Several of the fields are defined differently depending on which type of object is being defined. A *type* field identifies what kind of object is stored in a particular instance of the *tObject* structure. The objects are connected through pointers to form the linked structures used in *ClockWorks*. There are four different linked structures in *ClockWorks*.

This section will explain the important fields in the *tObject* data structure. This data structure is an important part of *ClockWorks*.

6.4.1 Name Information

Every instance of the tObject structure has five fields used for names.

- The *hName* field holds a unique identifier which is used to distinguish between instances of architecture objects. The *hName* field is always hidden from the user as it is used only by the Clock compiler.
- The *name* field holds the subview name of an event handler. Note that this subview name is the name of the subview being filled by the particular event handler class. The name field is not used for request handlers or iru objects.
- The *class* field holds the class name of a particular event handler or request handler. It also holds the name of an iru object.
- The *cName* and *cClass* fields hold the same information as the *name* and *class* fields. These two name fields are stored in a format which allows the names to be displayed on the screen. The *cName* and *cClass* fields are always kept consistent with the *name* and *class* fields.

6.4.2 Graphical Information

The fields of the *tObject* data structure which store graphical information are used to display the architecture structure in the work area of the main screen. The architecture structure is a tree which the user builds from the architecture objects available. The root of this tree is a global variable called *RealRoot*.

Each event handler in the tree stores the graphical information for all the objects in the node centered around it. However, each component sets its own width and height.

The height of an iru object is constant and its width varies with the length of its label (see figure 6.7).

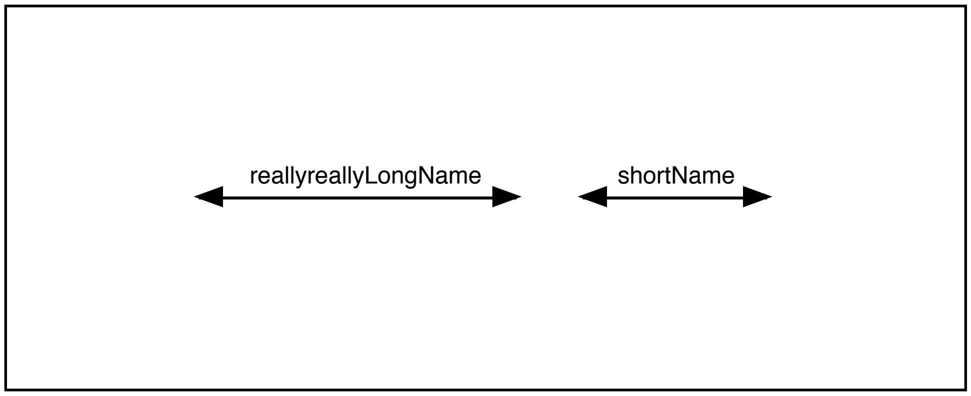


Figure 6.7 The Size of an input, request or update Object.

The width of a request handler is always the same as the width of the event handler it is attached to. When its interface is turned off, the height of a request handler is constant. However, when the interface is turned on, its height varies according to how many iru objects are attached to it. Each iru object attached to an event handler or a request handler, adds a constant amount of height to it. See figure 6.8 for an illustration of the sizes of request handlers.

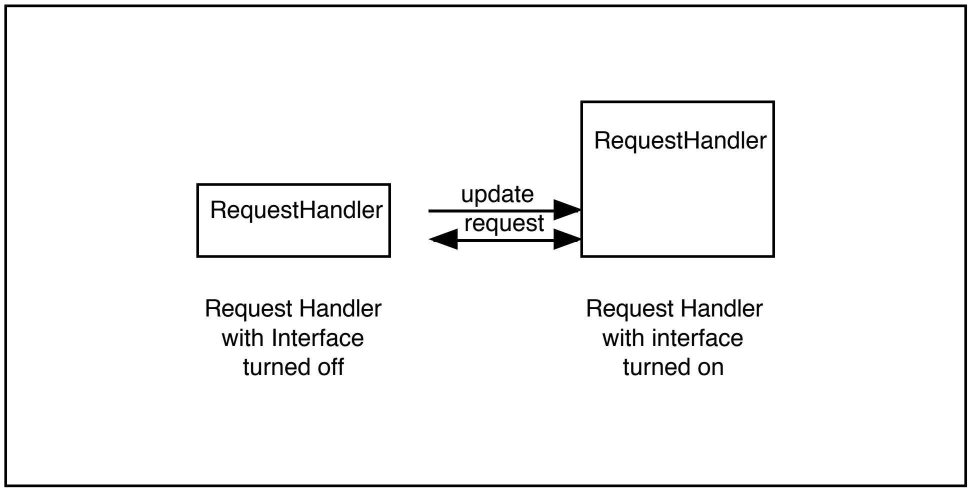

RequestHandler
update
request
RequestHandler
Request Handler with Interface turned off
Request Handler with interface turned on

Figure 6.8 The Size of a Request Handler Component.

The size of an event handler depends upon the sizes of the request handlers and iru objects which are attached to it. Please refer to figure 6.9 and the description below for an explanation of how the graphical information for an event handler (and a tree node) is stored.

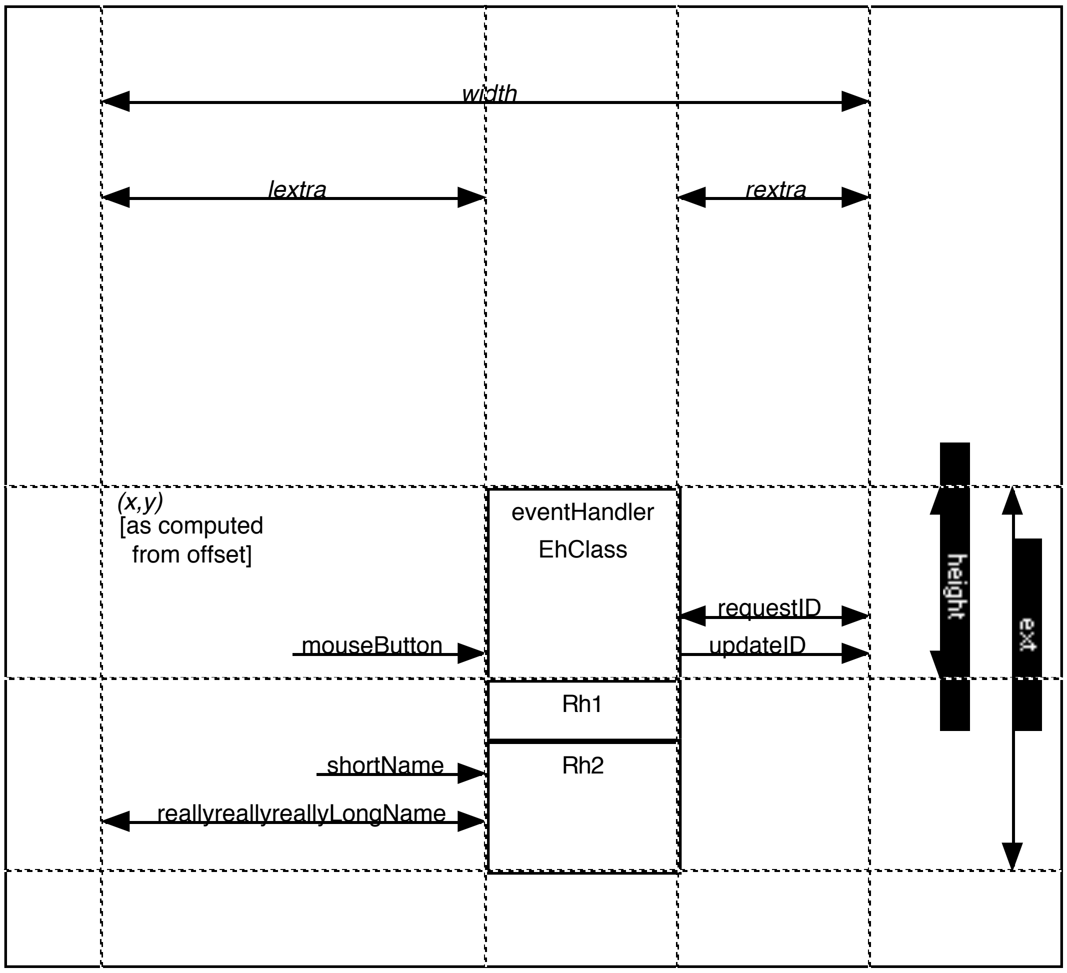


Figure 6.9 Size Computation for an Architecture Tree Node.

The width of the node has three parts, the width of the box, the width of the iru objects displayed to the left of the entire node, and the width of the iru objects displayed on the right side of the entire node. The *width* field of an event handler holds the sum of these three widths. This width is used in the layout algorithm to ensure that no tree nodes overlap each other on the screen. The fields *rextra* and *lextra* hold the width of the left and right irus respectively. The *rextra* and *lextra* fields are affected by all the iru objects currently displayed in the node which is based on the event handler.

The event handler and all the request handlers attached to it are represented as boxes. Each of these boxes has the same width. This width is stored in the event handler which forms the basis for the node. This width is set so that the box is wide enough to contain the longest name to be drawn inside the box.

The height of an event handler has 2 parts, the height of the event handler object and the height of the entire tree node which is based on the event handler. The height of the event handler depends upon the number of iru objects attached to it. Each iru attached to an event handler adds a constant amount to its height. The height of an event handler is constant if its interface is not turned on. The *ext* (extent) field of an event handler object holds the entire height of the node based upon the event handler. This value is the height of the event handler plus the height of each request handler attached to it. The *ext* field is used by the layout algorithm to ensure that the nodes do not overlap when the tree is drawn.

6.4.3 Pointers

The *tObject* data structure is used to build four different linked structures which are used by *ClockWorks*. Each structure is pointed to by a global variable. The variable *RealRoot* points to the architecture structure which is created by the user and displayed on the screen. The variable *tempEH* points to a list of event handler classes which have been defined for the current project. The variable *tempRH* points to a list of request handler classes which have been defined for the current project. The variable *tempIRU* points to a list of iru objects which have been defined for the project.

ClockWorks uses with a prototype instance model, which was explained in chapter 5. The model implies that the user works only with instances of classes. To modify a class, the user manipulates an instance of it. This is the model that the user sees. However, *ClockWorks* has to be able to clearly distinguish between a class and an instance of a class. Therefore, it keeps separate lists which contain the class declarations for each class defined by the user in the architecture structure. These lists are called the class declarations structures and they are hidden from the user.

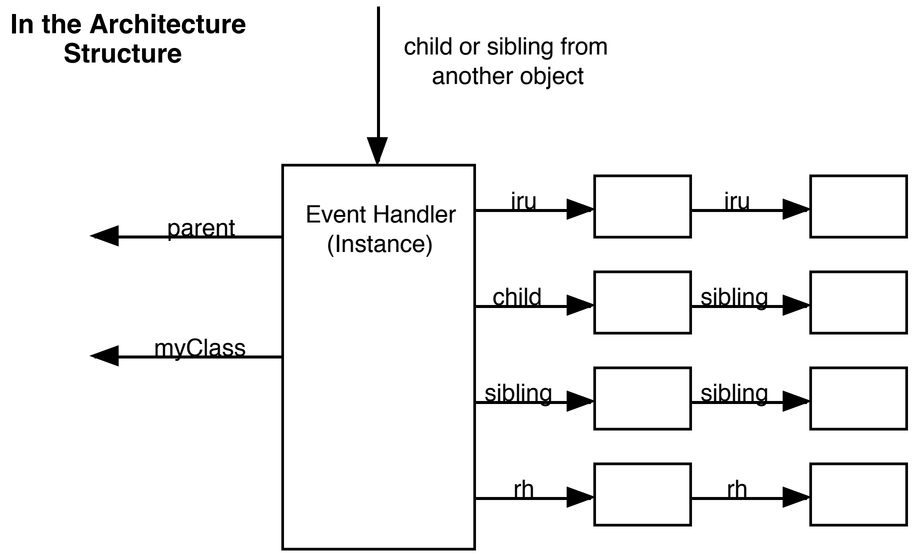
Each object has a number of pointers which connect it with the architecture structure and the class declaration structures. These connections are what puts an object into context and gives the architecture its meaning.

Objects in the architecture structure are connected to the class declaration structures through a pointer called *myClass*. Each object in the architecture sets this pointer to point to its class declaration in the class declaration structures. Whenever the user modifies a class through an instance of it in the architecture structure, the class is changed in the class declaration structure. This change is then broadcast to all instances of this class in the architecture structure.

Each object, whether in the class declaration structures or in the architecture structure, has the same pointers. These pointers are used differently for each type of object.

In the architecture structure, an event handler has a pointer to its *parent* and pointers connecting it with four linked lists. Figure 6.10 illustrates the pointer structure of event handler objects in both the architecture structure and the *tempEH* class declaration structure. The parent of an event handler is always an event handler. The *rh* pointer connects an event handler with the request handlers attached to it. This list is linked through the *rh* pointer. The *child* pointer connects the event handler with its children or subviews. This list is linked through the *sibling* pointer. The *iru* pointer connects the event handler with the *iru* objects attached to it. This list is linked through the *iru* pointer. The *sibling* pointer places the event handler in its position within its parent's list of children.

An event handler appearing in the class declaration structure for event handlers uses only two of these pointers. The *child* pointer connects the event handler declaration with a list of its subviews. This list is linked through the *sibling* pointer. The *iru* pointer connects the event hander declaration with a list of the iru objects attached to it. This list is linked through the *iru* pointer. Note that the child and iru objects attached to an event handler declaration do not actually represent objects. They are only used to store a list of the names of the children and irus attached to an event handler class.



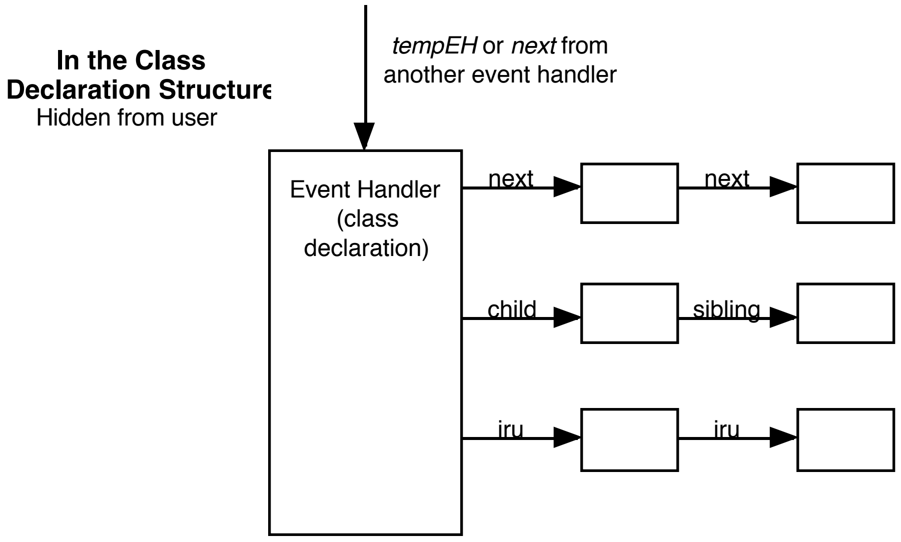

In the Class Declaration Structure
Hidden from user
tempEH or *next* from another event handler
Event Handler (class declaration)
next
next
child
sibling
iru
iru

Figure 6.10 Storage of Event Handler Objects.

In the architecture structure, a request handler (figure 6.11) uses its *parent* pointer to point to the event handler it is attached to. The *iru* pointer connects the request handler with the iru objects attached to it. This list is linked through the *iru* pointer. The *rh* pointer places the request handler within the context of its parent's list of request handlers. In the class declaration structure, the iru pointer points to a list of irus which are attached to this request handler class. This list is connected through the *iru* pointer.

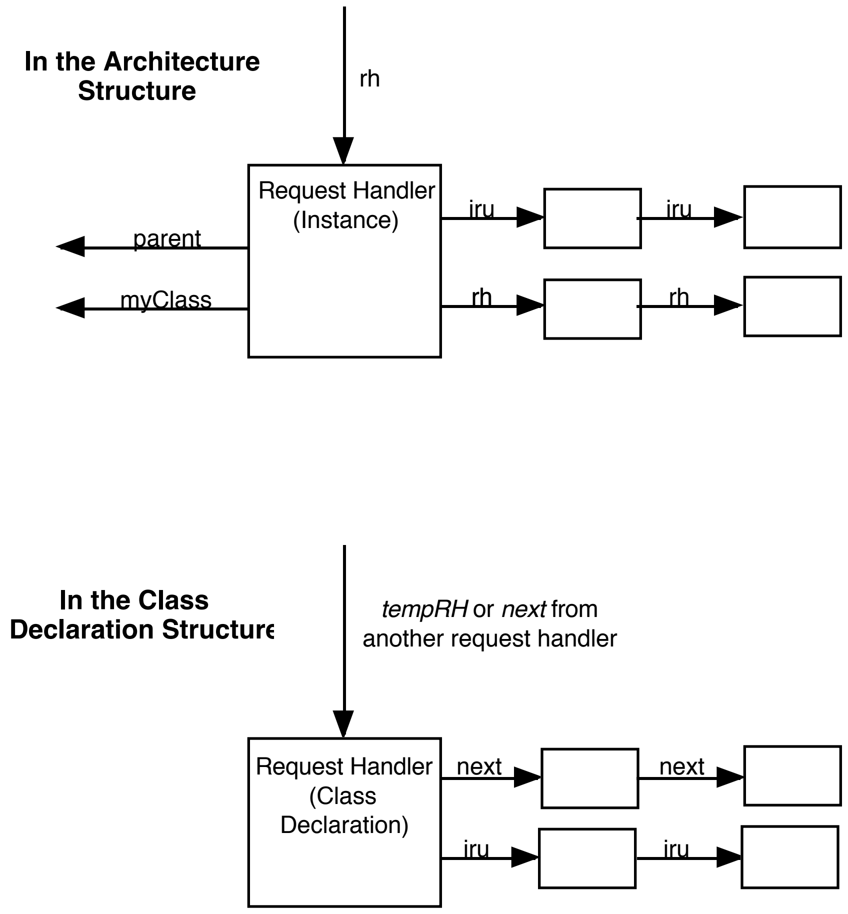

In the Architecture Structure
rh
Request Handler (Instance)
parent
myClass
iru
iru
rh
rh
In the Class Declaration Structure
tempRH or next from another request handler
Request Handler (Class Declaration)
next
next
iru
iru

Figure 6.11 Storage of Request Handler Objects.

An iru object (figure 6.12) uses its parent pointer to point to the request handler or event handler it is attached to. When an iru object appears in a class declaration structure, none of its pointers are connected. Note that Clock does not actually declare classes for iru objects. However, there cannot be two different iru objects with the same name, so a class declaration structure is required to ensure that all instances of an iru actually have the same information.

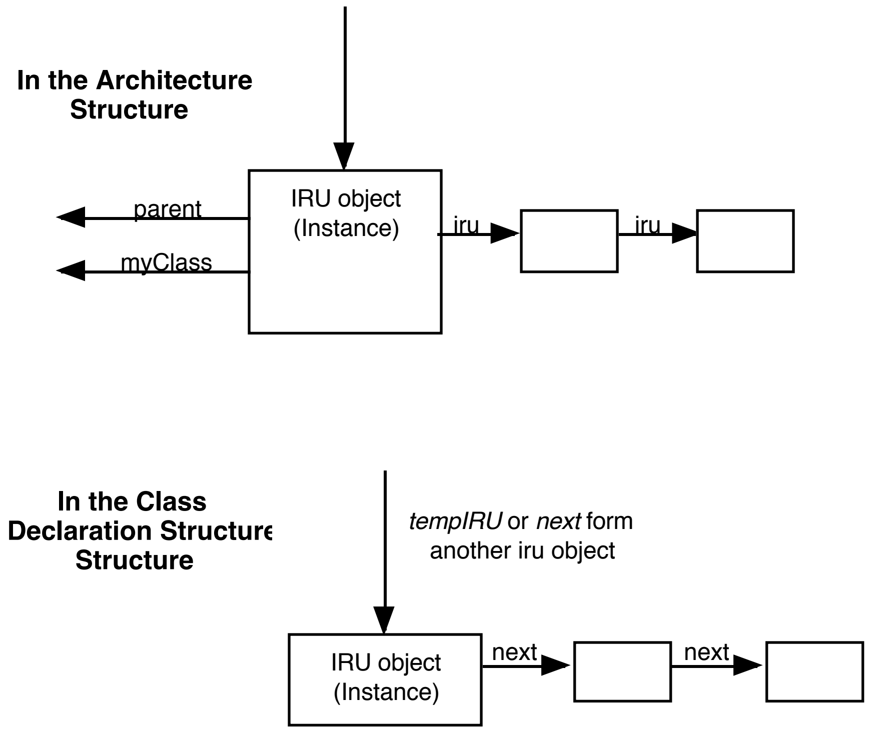

In the Architecture Structure
parent
myClass
IRU object (Instance)
iru
iru
In the Class Declaration Structure Structure
tempIRU or next form another iru object
IRU object (Instance)
next
next

Figure 6.12 Storage of Input, Request and Update objects.

6.5 Layout Algorithm

The layout algorithm used in *ClockWorks* was developed by Sven Moen [Moen 1990]. Generally, it computes an outline around each node and subtree and computes the position of each node so that they are as close together as possible, but still displayed as a tree. The way the tree is stored, each node points to a number of other components, but is only pointed to by one other node. The layout algorithm stores a node's position as an offset from the position of the node which points to it. Therefore, to draw the tree, *ClockWorks* simply traverses through each node, computes its position according to the given offset, and draws it on the screen. A node in the tree is displayed by drawing the event handler, the request handlers if they are to be displayed, and the iru objects if they are to be displayed.

6.6 Conclusion

The actual implementation of the first version of *ClockWorks* led to many interesting conclusions.

An experienced Clock user was able to play with *ClockWorks* and get a feel for its interface long before it was finished. This was due to the incremental style of implementation used. *ClockWorks* was almost always in an executable state.

The incremental implementation was encouraging because it was constantly being tested by one of the users who was very interested in seeing *ClockWorks* and anxious to use it to develop Clock applications.

The next chapter will explain the results of the user testing conducted for *ClockWorks*.

Chapter 7: Evaluation of *ClockWorks*

Component-based programming has the potential to be ideal for the creation of interactive applications. However, this potential cannot be realized without adequate program development support. Since the program architecture structures which form the basis of component-based applications are complex and difficult to work with, development tools for component-based programming must provide support for managing these structures. The development of *ClockWorks* was an experiment to demonstrate that the addition of a development tool designed specifically to support the creation and manipulation of program architecture structures could help overcome the difficulties associated with component-based programming.

The tasks and requirements identified for *ClockWorks* illustrated that developers had trouble understanding and working with their program's architecture structures. The resulting programming environment was built around the purpose of supporting user tasks specifically related to the comprehension and construction of these architecture structures.

This chapter presents the results of the evaluation and user testing of *ClockWorks*. The purpose of these evaluations was to determine how well *ClockWorks* met its goal of supporting the task of managing the complexity of Clock architecture structures. More specifically, the goal of the formal user testing was to evaluate how well *ClockWorks* supported the requirements identified in chapter 4. Version one of *ClockWorks* successfully supported the first seven of these requirements, the last three will be supported in a future version.

There has been some form of evaluation performed at each stage of the development of *ClockWorks* to ensure that the requirements were properly supported.

The user interface design (described in chapter 5) was demonstrated to the users through sketches and examples. It went through several modifications in response to user feedback. The final evaluation of the user interface design was a cognitive walk-through to locate redundancy, unnecessary complexity, or missing tasks. These evaluations are explained in detail in section 7.1

The incremental implementation of *ClockWorks* (described in chapter 6) allowed users to work with the various commands and features as they were added. In this way, the commands and interface was refined and modified as its development progressed. This evaluation is described in section 7.2.

The Interactive Calendar (described in chapter 2) was re-implemented with *ClockWorks* to make a comparison between the original Clock development support and *ClockWorks*. This is described in section 7.3.

Once the first version of *ClockWorks* was complete, formal user testing was performed to determine whether or not it succeeded in its goals. In general, the results of user testing have shown that the first seven requirements identified in chapter 4 have been successfully supported in *ClockWorks*. All the users surveyed commented that they will use *ClockWorks* to develop Clock programs in the future. The results of the formal user testing are explained in detail in section 7.4.

7.1 Evaluation During Requirements Specification and Design

It is important to evaluate the user interface design before implementation begins. Evaluation will help in the discovery of missing task support or missing detail in the design. It will also give the users another opportunity to participate in the design. Without evaluation these details might not be discovered until implementation is begun and modifications would be costly.

ClockWorks underwent two levels of testing before any implementation was begun. The design described in chapter 4 is the final design of *ClockWorks* after all of these evaluations took place.

7.1.1 User Evaluation

The user interface design of *ClockWorks* was explained to the users through sketches and examples. Comments from the users provided general ideas about how the design could be improved.

The initial version of the user interface design was far more complex than the final version. Some necessary details about certain aspects were not included, some features were not intuitive or overly complex. Through user feedback, *ClockWorks* was made more precise and easier to use. These discussions with the users were also very helpful in gaining a better understanding of the Clock system and the way its users work with it.

The design underwent many changes throughout the period of user evaluation. This design-evaluation loop went through several iterations before the user interface design was subjected to more rigorous evaluation. The user evaluation provided the feedback necessary to decide whether or not the design was feasible and whether it was ready to implement.

7.1.2 Task Oriented Specification

The user feedback did not give enough precise information to determine whether or not the user interface design would support all of the tasks indicated in the user needs analysis.

The Task Oriented Specification phase of the Clock Methodology (see chapter 2) provided the necessary detailed analysis. In this step, the user interface was formally evaluated by representing each of the tasks identified in the task analysis in a language called User Action Notation (UAN) [Hartson *et al*. 1990, 1992]. The result of this step was a detailed, formally defined description of how each user task could be carried out with *ClockWorks*. This was all within the context of the original task hierarchy. The UAN descriptions of these tasks are included in Appendix B.

The task of performing the UAN description of the user interface provided a useful idea of what it would be like to use *ClockWorks*. It made redundancy and unnecessary complexity obvious. Also, it clearly illustrated that there were some tasks which were not supported well or were not supported at all in the design.

The UAN description helped to get the user interface design ready for implementation. Problems with the user interface design could be clearly seen once it was represented this way. The added information included with the UAN helped in planning how the implementation would be carried out. It provided important details such as interface feedback and internal state. These details may have been overlooked in the initial design, but they were necessary once the implementation was started.

While the user evaluation provided general opinions and ideas about the user interface design, the task oriented specification forced more specific detail into the design in preparation for implementation.

7.2 User Evaluation during Implementation

ClockWorks was implemented incrementally so that it was almost always in an executable state. One of the users worked with the partially-completed program to give immediate feedback about each command as it was implemented. This user feedback can be grouped into three categories:

Comments in the first category related to the user interface. The menu commands were renamed and reorganized in response to user comments. Most problems with the user interface were minor and were easily overcome.

The second category of comments were suggestions of new features or commands which could be added. The ideas for these additions came from using *ClockWorks* and recognizing things which would make it more powerful or better to use. For example, the commands to rename, replace or duplicate an object were not part of the original task analysis or the initial list of requirements. These commands were added during implementation. Another new feature was the ability to hide local or internal input, request and update events attached to an event handler when its request handlers were hidden. This helps the user manage the complexity of the architecture structure.

The third category of user comments during implementation were reports of bugs and commands which were not working properly. These did not involve any design modifications. However, these reports were very helpful in eliminating bugs from the program.

Over all, user comments throughout implementation resulted in several important modifications to the user interface design of *ClockWorks*.

7.3 The Interactive Calendar Program

The first part of the final testing of was to implement the interactive Calendar program (described in chapter 2) with *ClockWorks*. The architecture structure of this application was completely redone with the *ClockWorks* although the same code was used to define each component.

This test had two goals: first, it was to ensure that the *ClockWorks* was as bug-free as possible, and second it was an experiment to compare the old style of developing a Clock program with the new style defined by *ClockWorks*.

The Interactive Calendar was the first application to be built with *ClockWorks*. The definition of the architecture structure took very little time. The graphical representation of this architecture was so easy to understand that it clearly identified some problems with the initial architecture structure. The architecture was simplified and improved.

In general, the entire process of developing a Clock program has been simplified by *ClockWorks*. It is now much easier to see the potential of the Clock system for the creation of interactive applications. The importance of the architecture structure is more evident. It is easier to create a complete and correct architecture before writing the code for the components. There is less chance of discovering architecture errors during compilation and execution. However, in the event that these errors occur, they are much easier to find with

ClockWorks. The graphical representation of the inputs, requests, and updates provides an easy way to verify the architecture structure and locate problems. The implementation of the Interactive Calendar program clearly illustrated the benefits of *ClockWorks*.

7.4 Formal User Testing

The first version of *ClockWorks* was tested using a set of guidelines for user observation and testing [Gomoll and Nicol 1990]. Before user testing began, a group meeting was held in which the first version of *ClockWorks* was demonstrated. The users were also asked to read the documentation for *ClockWorks*.

In preparation for the test, a small Clock application was designed and partially implemented. The Coins application was designed to count loose change (quarters, nickels, dimes and pennies). The initial version of this application is illustrated in Figure 7.1 and its architecture structure can be seen in figure 7.2. The incomplete program given to the users consisted of four numeric fields. There were two buttons associated with each field - one to increase the number and one to decrease it. Each field corresponds to a different kind of coin. The users were asked to complete this application by adding two new features. The first new feature was to add functionality to keep track of and display the total amount of money (in cents). For instance, if there are 2 quarters, 1 dime, and 3 nickels, and 2 pennies then the total should be 77 cents. The second new feature was to add a reset button to reset all numeric fields to 0. Figure 7.3 is an example of the completed application. The architecture of the completed application is displayed in figure 7.4.

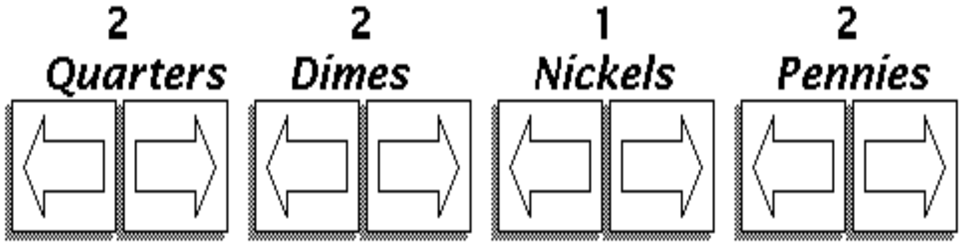


Figure 7.1 The Incomplete Coins application. The number of each type of coin is increased or decreased by pressing the arrow buttons below it.

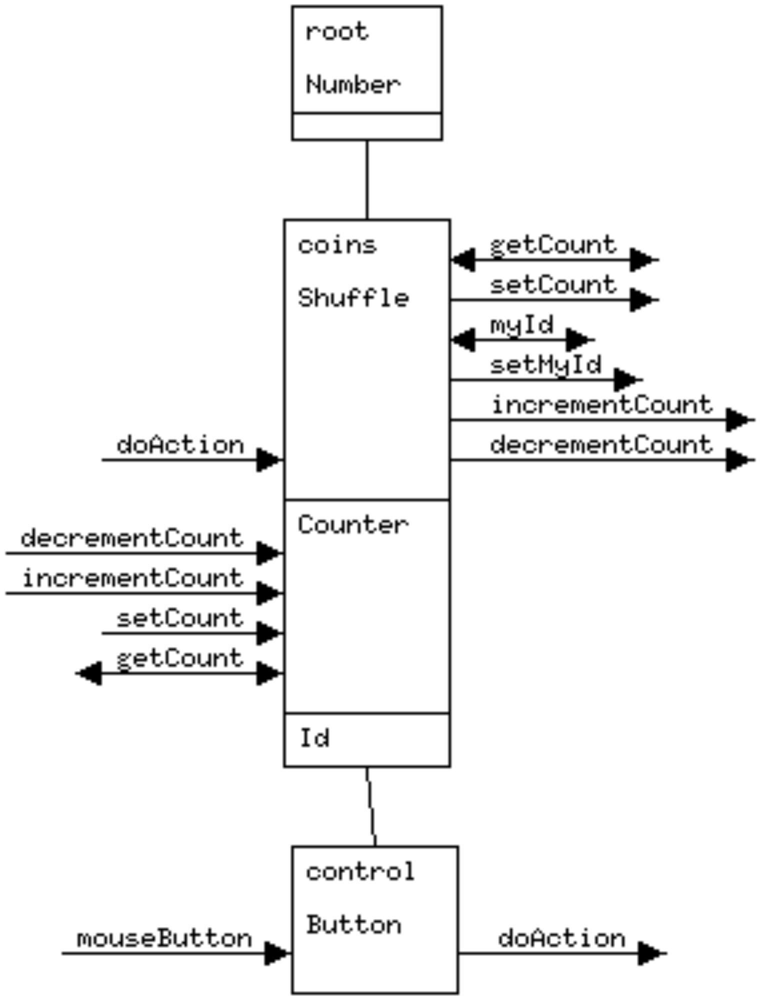


Figure 7.2 Architecture of the incomplete Coins application. The top event handler Number *creates four instances of its subview* coins. *Each subview is a* Shuffle *event handler set up to increase or decrease a* Counter *in response to the* doAction *input from the* Button *event handler.*

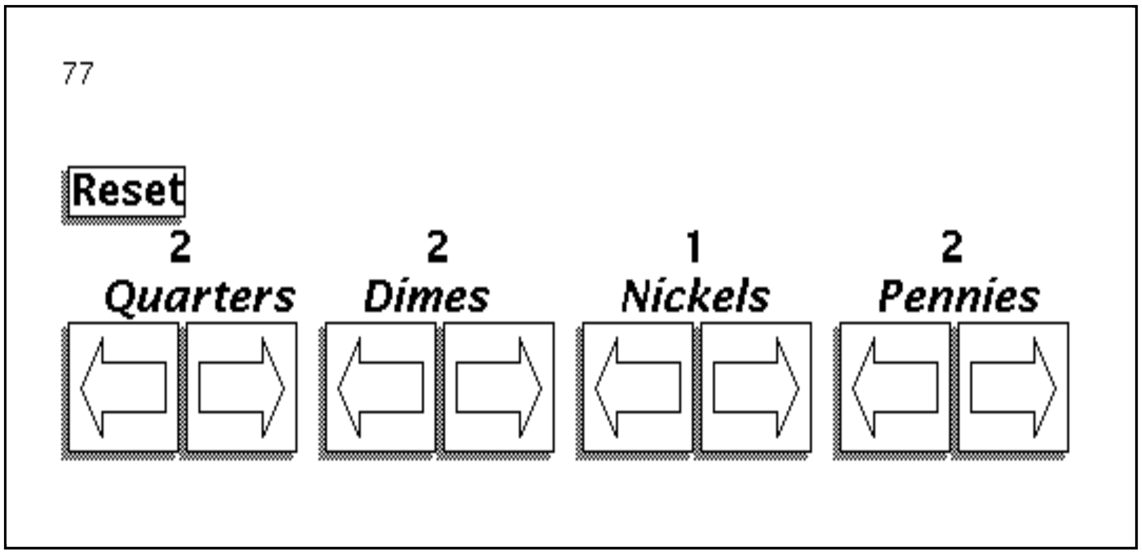


Figure 7.3 The complete Coins application after the total *and* reset *features have been added*.

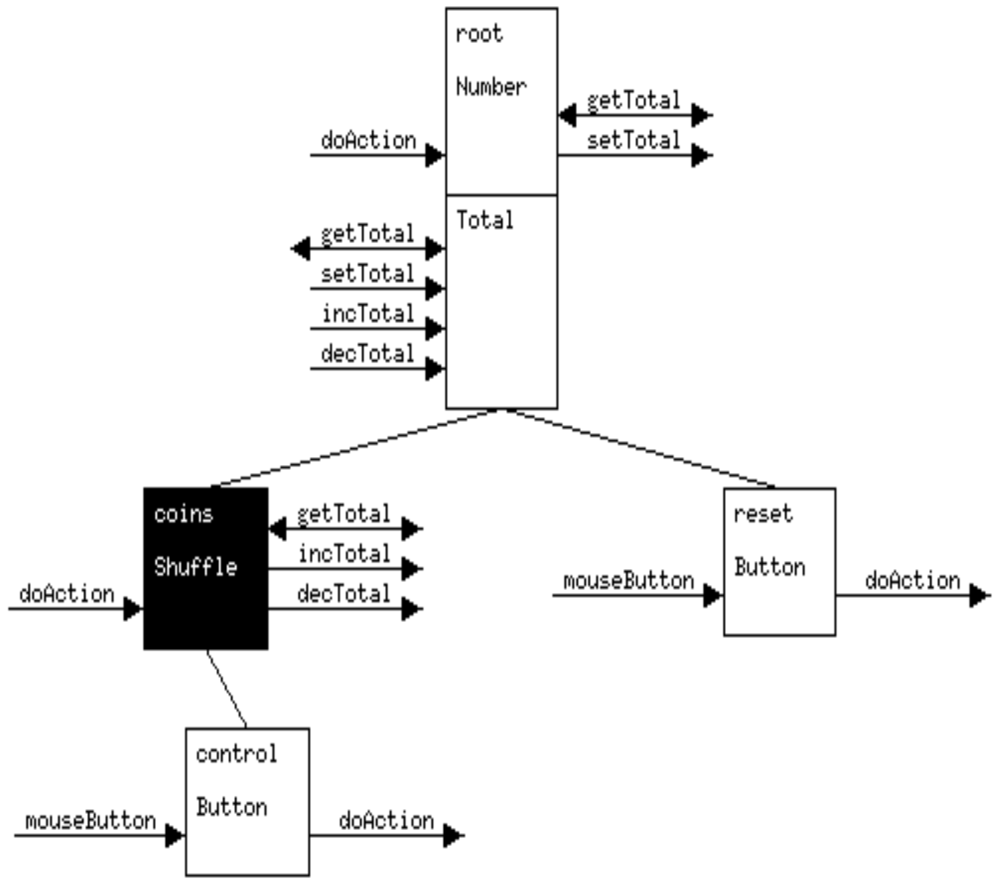


Figure 7.4 The architecture structure of the complete Coins *application. The new* reset *subview added to the* Number *event handler controls the* reset *feature. The new request handler* Total *is updated each time the user changes the number of any type of coin.*

Tests were conducted individually. Each user was given an hour to work on the tasks. The modifications to the application were made with *ClockWorks*. The purpose of this experiment was to observe the users interacting with *ClockWorks*. The users were asked to 'think aloud' so that their interactions with *ClockWorks* could be understood clearly.

The tasks were chosen so that they would require the addition of at least one of every type of architecture object (request handler, event handler, input, request, and update). Also, since the tasks involved modifying an existing application rather than starting a new one, the users had the added task of understanding an existing architecture structure.

The formal user testing was carried out after the first version of *ClockWorks* was completed. Its goal was to see how well users were able to interact with the system and to determine how well the requirements were supported.

7.4.1 User Observation

The four users who participated in the formal user testing had experience using the original Clock system. Of these, one had experience using *ClockWorks* throughout its development, and three had only been briefly introduced it as preparation for the user testing.

Several conclusions about *ClockWorks* and its design were drawn after observing the users and listening to them 'think aloud' while carrying out the assigned tasks.

ClockWorks stresses the importance of the architecture structure. The users were not accustomed to this emphasis. Formerly, Clock developers were forced to find their own method for understanding this architecture structure. Most developers made use of a hand-drawn architecture picture, but the level of architecture detail represented on these pictures varied for different users. Some users chose not to draw inputs, requests, and updates as part of the architecture structure. They had some trouble associating these objects with the architecture structure in *ClockWorks*. The initial reaction of these users was to open the editor for each component and try to understand its interface with other components by reading its functional code definition. However, since *ClockWorks* represents all the inputs, requests, and updates within its proper context in the architecture structure, there is no need to open the code for any component to understand how it interacts with other components. It is possible to fully comprehend an architecture structure without looking at any code.

Initially, three of the four users opened the code for components before realizing that the information could be displayed in the architecture picture. Of these, two quickly realized that it was not necessary to use the code to understand components. They quickly closed the editors and used the interface detail to understand the architecture. These three users went on to make all the necessary changes to the architecture structure before modifying any Clock code.

One person used the code to understand and modify the interface for each component without using interface detail contained in the architecture structure. This led to compilation problems caused by the fact that the architecture definition did not match the code for each component. In the original Clock system, the architecture structure was so difficult to manage that developers often just went directly to the coding stages for each component once the initial structure of event handlers and request handlers was defined. Once the code was defined, these developers went back and filled in the interface portion of the component class declarations to correspond to the code for each component. It is interesting to note that this user completed fewer modifications to the application than the other users in the same amount of time.

ClockWorks gives developers a very structured way to create Clock applications. It therefore takes some time for developers to become accustomed to the new environment if they used a different method for defining their programs with the original system.

The three users who had only a small amount of experience with *ClockWorks* were slow getting started. This was caused by the fact that *ClockWorks* was new to them and by the fact that they had not used the Clock system for several months. Part of their confusion was remembering how Clock architecture structures functioned. In all cases, once they had used *ClockWorks* for a short time, they were able to proceed with their modifications with little trouble.

From the observations of users interacting with *ClockWorks*, it became clear that some minor modifications to its user interface would make it easier to use and to learn.

Every user experienced some confusion with the *Open* and *Close* commands which are defined to open or close a new level of detail for a component. In a future version, this feature could be made less confusing if it was replaced by individual commands to open or close each level of detail. The i*nterface/hide interface* command could be used as a model for a collection of new commands: *hide/show request handlers*, and *hide/open sub-tree*. A separate command could be used to invoke the editor.

Most users commented that the process of selecting a component then selecting a command from one of the menus was a burden. They would like to see a way to make this process faster. A new version might include a way to access various commands for the components in a single action.

The semantic feedback used to indicate the level of detail displayed for components was not useful. For instance, three of the users did not understand the small rectangle displayed under and event handler to indicate that it had no request handlers. *ClockWorks* did not contain any semantic feedback to indicate that components had no interface. This caused some confusion as users repeatedly selected the show interface command thinking that it did not work the first time. *ClockWorks* also provides no indication that the editor for a component is open. In a future version, color or graphics could be used to indicate such information.

In general, the users were successful in carrying out the assigned tasks with *ClockWorks*. They were able to understand and modify the Clock application. Two of the four users completed all the required tasks. One completed all the architecture modifications but didn't modify the code, and one completed half of the required tasks before running out of time.

After using *ClockWorks* for a short time, the users were able to use it without any outside help. None of the users used any tool other than *ClockWorks* to perform the tasks nor did they make use of any hand drawings or sketches of the architecture structure. The next section provides a summary of the users' reactions to *ClockWorks*.

7.4.2 Final User Feedback.

Once the users had completed the tasks specified in the formal user test, they were asked to comment on how well each requirement was supported by *ClockWorks*. The users agreed that the first seven requirements identified in chapter 4 were supported by *ClockWorks*. There were some minor complaints about the user interface. Specifically, there should be some improvement in the commands used to hide and display the detail of the various components. In general, the comments made by the users were similar to the conclusions drawn from the user observation. They can be summarized into three categories:

The first category includes small changes to the interface which would make it easier to use and to learn. These comments included minor problems with the user interface of *ClockWorks* which could be modified easily.

The second category includes comments about requirements which were not supported in *ClockWorks*. Most users felt that *ClockWorks* would be even better if it supported the last three requirements identified in chapter four. These are: support for architecture verification, support for component grouping, and simple code generation. These and other features will be added to a future version of *ClockWorks*.

The third category includes comments which were related to the Clock compiler which is currently being re-written. *ClockWorks* only supports tasks relating to the creation and manipulation of the architecture structure of Clock programs. Future versions will be integrated with the new Clock interpreter when it is finished. When this modification is made, messages from the Clock system and the interpreter will be displayed in a scrolling window under the architecture drawing.

7.5 Conclusions

Generally, once user testing is complete, the new information should be used to modify the user interface design of an application so that the problems uncovered can be overcome. Modifications to version one of *ClockWorks* were not made as part of this thesis due to lack of time.

The graphical representation of the architecture structure provides a clear understanding of the structure of Clock applications. This allows programs to be defined easily, and it also helps developers create better and more efficient architecture designs and improve existing designs.

The ability to comprehend architecture structures has several implications. Architecture structures can be easily modified by adding or replacing components. Pre-defined components are represented in such a way that their interface can be understood. This makes their integration into the architecture structure easier and more convenient.

ClockWorks can be used to build components and add them to the library. Once an adequate library of interaction techniques has been defined, a Clock application can be built entirely from pre-defined components. Once the group components have been added to the Clock system, software reuse will be even more convenient.

In general, the users felt that *ClockWorks* made the development of Clock Programs easier. The architecture structures are easier to understand and modify. Every user tested said they would rather use version one of *ClockWorks* than the original Clock system. This proves that *ClockWorks* was successful in its goal making the Clock system easier to learn and to use.

Chapter 8: Summary and Conclusions

8.1 Summary

Research has shown that the component-based style of programming has the potential to overcome some of the difficulties often associated with the development of interactive software and software reuse. However, component-based systems are too difficult to use if they do not provide tool support to help developers manage the complexity of the architecture structure of their programs. These architecture structures define how the components are joined to form an application. Although they provide a high-level structure for the program, they also contain a great deal of information, all of which is important to the a proper understanding of the application being built. Without this understanding, developers were be unable to deal with large applications, or make modifications to existing applications.

Clock is a component-based programming system in which components are defined using a functional language. Like many other component-based systems, Clock defines a model upon which its applications are structured. Clock components are connected into a hierarchical structure which breaks an interactive application into its individual components. The Clock system has many advantages for the development of interactive applications. Once the architecture structure has been defined, the individual components can be easily defined with very little code. This feature makes Clock an ideal tool for rapid prototyping. Modifications to applications can be made by modifying or replacing individual components. A library of pre-defined components can be created to allow programs to be built with very little coding.

The disadvantage of the Clock system was that it had no effective development tools to support the creation and manipulation of program architecture structures. Developers were not able to take advantage of the benefits of the Clock system because it was too difficult to manage the architecture structures of programs. These structures very quickly became too large and complex to manage.

A task analysis of the Clock programmer identified lack of tool support as the root of its problems. A set of requirements for program support was identified and *ClockWorks* was designed. The first version of *ClockWorks* has been implemented and tested with a group of Clock programmers. Results of this testing indicate that *ClockWorks* has overcome the problems originally associated with the Clock system.

8.1 Implications

ClockWorks provides graphical representation and direct manipulation of Clock architecture structures. This allows programmers to easily understand and manage the complexity of their program architectures. The Clock system can now be used to create larger applications. Also, applications developed with the Clock system can now be designed to be more efficient. The graphical representation of the architecture structure clearly identifies problems or redundancies in the architecture structure. The completion of the first version of *ClockWorks* has had the following implications:

- Developers can display the architecture structure of their programs in a way which will be intuitive and easy to follow no matter how large their application becomes.
- The graphical representation of the architecture structure gives programmers a better understanding of how the Clock system works. This makes the Clock system easier to learn. It also helps developers create better and more efficient Clock applications.
- *ClockWorks* makes it easier for developers to find errors in the architecture structure of their applications. Once support for architecture verification has been added to a future version of *ClockWorks*, developers will have less difficulty creating correct architecture structures.
- *ClockWorks* allows developers to give the design of their architecture structures more consideration. Developers can now modify their architecture structure easily. Developers will be able to make modifications to their program design in response to user feedback. The resulting applications will provide better support for user tasks.
- *ClockWorks* allows applications to be developed incrementally. Developers can create rapid prototypes of applications or specific portions of applications. These prototypes can be tested and refined as the application is constructed. This gives developers of interactive software the ability to modify and refine applications in response to user feedback. The resulting applications will more precisely support user tasks. Rapid prototyping and incremental implementation will be even more convenient once support for code generation has been added to *ClockWorks*. With this feature, developers will be able to create executable prototypes without writing any code.
- *ClockWorks* assists developers in designing components which can easily and conveniently be reused. Also, it makes the process of reusing a component more convenient. The graphical representation of each component clearly indicates the interface through which it communicates with other components. This information makes it easier for a developer to connect a pre-defined component to an existing architecture. Once support for group components has been added to *ClockWorks*, developers will be able to reuse entire architecture structures or sub-trees.
- *ClockWorks* provides graphical support to assist developers in managing the complexity of the architecture structure of applications built using a component-based programming system. Since component-based programming systems stress the importance of the structure of components connected to build an application, the ideas used in the creation of the Clock programming environment could be adapted for use with other component-based systems.

8.3 Conclusions

The goal of this thesis was to define a software development environment to help developers of component-based applications manage the complexity of their program architecture structures. *ClockWorks* is a tool which allows developers to build and manipulate the architecture structures which define Clock applications. The first version of *ClockWorks* has been implemented and tested with its users. Conclusions have shown that it has managed to overcome many of the problems which were associated with the ability to manage Clock architecture structures.

Although *ClockWorks* was developed specifically for the Clock system, the techniques and ideas used for its design and implementation are applicable to other component based systems. The main similarity between Clock and the other component-based systems studied was that applications are built as structures of software components.

The components are defined separately and structured to form an application. *ClockWorks'* support for the creation and manipulation of Clock architecture structures could be adapted for use with other component-based systems.

Care has been taken to ensure that the environment met the needs of its users. The results of the formal user testing have shown that seven of the ten requirements identified in chapter 4 have been met.

8.4 Future Plans for the Clock System

The following list indicates some of the future plans for improving the Clock System:

- A graphical editor is currently being developed which will allow developers to mix textual and graphical languages to specify Clock view functions [Song 1994]. This tool will be integrated in a future version of *ClockWorks*.
- A future version of *ClockWorks* will add support for code generation. With this feature, default code will be generated when components are defined.
- A future version of *ClockWorks* will include support for architecture verification. *ClockWorks* will automatically verify input, request and update events when they are added to the architecture. Semantic feedback will be used to notify the developer when the source of destination of such events cannot be resolved.
- A future version of the Clock system will add support for group components. These components will be used to represent architecture trees or sub-trees as a single component. This feature will make software reuse more effective and convenient. These changes will also be added to a future version of *ClockWorks*.
- An interpreter for Clock functional language is currently being written. It is hoped that this interpreter will help overcome some of the performance problems currently associated with the Clock system.

References

[Booth, 1989]
Paul Booth and Chris J. Marshall. “Usability in Human-Computer Interaction” from *An Introduction to Human-Computer Interaction*. Lawrence Erlebaum Associates, 1989. pp. 103-117.

[Carey *et al*. 1989]
M.S. Carey, R.B. Stammers, and J.A. Astley. “Human-Computer Interaction Design: the Potential and Pitfalls of Hierarchical Task Analysis.” from *Task Analysis for Human-Computer Interaction* edited by Dan Diaper. New York: Halsted Press, 1989. pp. 56-73.

[Coutaz, 1987a]
Joelle Coutaz. "The Construction of User Interface and the Object Oriented Paradigm." *The European Conference on Object Oriented Programming*, 1987. pp. 135-144.

[Coutaz, 1987b]
Joelle Coutaz. "PAC, an Object Oriented Model for Dialog Design." *Human-Computer Interaction - INTERACT* 1987. pp. 431-436.

[Coutaz, 1989]
Joelle Coutaz. "Architecture Models for Interactive Software." *The European Conference on Object Oriented Programming*, 1989. pp. 383-399.

[Damker 1992]
Herbert Damker, "Spezifizierung und Architekturentwurf von Benutzungsschnittstellen: eine Fallstudie in der Clock-Methodologie", University of Karlsruhe, Germany, 1992.

[Gomell 1990]
Kathleen Gomell. “Some Techniques for Observing Users.” in *The Art of Human-Computer Interface Design*. edited by Brenda Laurel. Addison-Wesley, 1990. pp. 85-93.

[Graham 1992]
T.C. Nicholas Graham. “Constructing User Interfaces with Functions and Temporal Constraints.” in *Languages for Developing User Interfaces*. edited by Brad A. Myers. Jones and Bartlett, 1992. pp. 279-302.

[Hartson *et al*. 1990]
H. Rex Hartson, Antonio C. Siochi and Deborah Hix. "The UAN: a User Oriented Representation for Direct Manipulation Interface Designs." *ACM Transactions on Information Systems*. Volume 8, Number 3. July 1990. pp. 181-203.

[Hartson *et al*. 1992]
H. Rex Hartson, Jeffrey L. Brandenburg and Deborah Hix. "Different Languages for Different Development Activities: Behavioral Representation Techniques for User Interface Design" in *Languages for Developing User Interfaces*, edited by Brad Myers. Jones and Bartlett, 1992. pp. 303-328.

[Hartson and Hix 1989]
H. Rex Hartson and Deborah Hix, "Human-Computer Interface Development: Concepts and Systems", *ACM Computing Surveys*, volume 21, number 1, March 1989. pp. 5-92.

[Hill 1992]
Ralph D. Hill. "The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications" *ACM CHI* 1992. pp. 335-342.

[Jeffries *et al*. 1991]
Robin Jeffries, James R. Miller, Cathleen Wharton, and Kathy M. Uyeda. “User Interface Evaluation in the Real World: A Comparison of Four Techniques”. *ACM SIGCHI 1991*, April, 1991. pp. 119-124.

[Krasner and Pope 1988]
Krasner, Glen E. and Stephen T. Pope. "A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80." *Journal of Object Oriented Programming*. August/September, 1988. pp. 26-49.

[Mancoridis, Holt, Penny 1993]
Spiros Mancoridis, Richard C. Holt, David A. Penny “A ‘Curriculum-Cycle’ Environment for Teaching Programming” ACM SIGCSE 1993. pp. 15-19.

[Moen 1990]
Sven Moen. “Drawing Dynamic Trees”. *IEEE Software*, July 1990. pp. 21-28.

[Moriconi and Hare 1985]
Mark Moriconi and Dwight F. Hare. "PegaSys: A System for the Graphical Explanation of Program Design" *ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*. July 1985. pp. 148-160.

[Myers *et al*. 1990]
Brad A Myers *et al*. "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*, November 1990. pp. 71-85.

[Myers 1992]
Brad A. Myers. "Ideas from Garnet for Future User Interface Languages" in *Languages for Developing User Interfaces* . edited by Brad Myers. Jones and Bartlett, 1992. pp. 147-159.

[Myers 1993]
Brad A. Myers. “Why are Human-Computer Interfaces Difficult to Design and Implement?” Technical report CMU-CS-93-183. Carnegie Mellon University, 1993.

[Myers and Rosson 1992]
Brad A. Myers and Mary Beth Rosson “Survey on User Interface Programming”. *ACM CHI* 1992. pp. 195-202.

[Nierstrasz *et al*. 1992]
Oscar Nierstrasz, Simon Gibbs and Dennis Tsichritzis. “Component-Oriented Software Development”. *Communications of the ACM*. September 1992. Volume 35, No.9. pp. 160-165.

[Penny 1993]
David Allen Penny. "The Software Landscape: A Visual Formalism for Programming in the Large." University of Toronto, 1993.

[Reiss 1987]
Steven P. Reiss. "Working in the Garden Environment for Conceptual Programming" *IEEE Software*, 1987. pp. 16-27.

[Song 1994]
Gekun Song. “Mixing Visual and Textual Programming in Functional Languages.” Master’s thesis, York University, North York, Canada, August 1994 (expected).

Appendix A: Task Analysis of the Clock Programmer

This appendix contains the complete description of tasks created as part of the task analysis described in chapter 4. The task hierarchy corresponds to figures 4.2, 4.3 and 4.4.

Main Goal: To develop an interactive application.
The entire development of an interactive application is an iterative process. Some users choose to work on the entire application design and implementation, while others design and implement the various components separately.

> **Sub-Goal: User Needs Analysis**
> The user needs analysis is an evaluation of users and their needs. Each task currently performed by the user is analyzed in order to determine which will be addressed by the application to be developed.
>
> **Sub-Goal: User Interface Design**
> The result of this sub-goal is a drawing or prototype of the user interface of the application to be developed.
>
> **Sub-Goal: Task oriented specification**
> This sub-goal analyses the UI design with respect to the user and the tasks indicated in the user needs analysis.
>
> **Sub-Goal: Architecture Design**
> This sub-goal is to produce a complete, architectural design for the application to be developed.
>
> > **Sub-Goal: Design tree structure**
> > This sub-goal makes use of the UI design completed earlier. The tree structure is a hierarchical model of the components which make up the application and the way in which they communicate.
> > The application's architecture structure is initially designed on a large piece of paper. This hand drawing becomes cluttered as each level of detail is added and as modifications are made. It can become difficult or impossible to follow. One solution to this problem is to work only on a small part of the tree at a time. Sections of the main diagram can be separated thus allowing more detail to be added to the hand-drawn architecture design. Some users chose not to add certain levels of detail to their architecture design at all. Instead, this portion of the design is done as they begin to create the declaration files for the various components. However, for large applications, this is not a feasible solution. Some representation of this detail is needed in order to understand the design once it has been created.

This understanding is necessary for modifications or additions to the original design which are made frequently.

Task: Add an event handler:

An event handler represents the behavior and appearance of some functional component of the program being designed. It may be made up of a collection of event handlers. The entire application is represented as a hierarchical structure of event handlers based on the UI design sketches. Lower levels represent smaller details. The highest level (root) represents the entire application.

There are no concrete rules which defines exactly how the component structure should be arranged or which components should be broken down into smaller components. However, there are some guide-lines:

In general, any part of the UI which performs some function can be considered a separate component. Text labels are not considered separate components because they don't perform any functions in the application. However, buttons have a specific function, so they are always separate components.

If there is a part of the UI which appears and functions as a separate component, then it should be a separate component with its detail one level down the tree. For example, a scrollbar is a separately functioning part of an application. The component called scrollbar is further divided into its own functional components: the arrow buttons and the sliding thumb.

An event handler can be used to group a collection of child components of the same class. For example, a menu of commands is represented as a list of buttons. A component called *Actions* is the parent and the various buttons are its children.

Task: Add a request handler

A request handler is used to hold the data needed by the application. It handles both requests for the data it holds as well as updates to the data. A request handler must be placed above the components which depend on it. It should be placed as low as possible in the tree to avoid excessive redrawing while the application is running.

There is no specific method defined to indicate how request handlers should be added to the tree. There are several possible ideas to give guidance to this task.

Some request handlers come from the connection to computation column of the UAN description generated in the task oriented specification sub-goal of system development. Most such request handlers are placed at a high level in the architecture tree because they are closely related to computation.

Request handlers pertaining to the state of the UI generally can be related to the 'connection to computation' column of the UAN description.

Components or sub-trees of components which are being reused from another application will have the same or similar request handlers as they did in their original application. These request handlers help to define the functionality of a component. For example, a button always has an *Id* to identify the function it performs, and a *Status* which indicates the state of the button (either selected or unselected).

Task: Design flow of inputs

Inputs are handled by event handlers. They either come from the user or from a lower-level event handler.

Working from the lowest level in the tree, inputs can be drawn onto the hand drawing of the architecture structure by connecting event handler components or indicating user input with arrows or labels. Some users do not add this level of detail to their hand-drawing.

Input events from the user include mouse button, or keyboard activity. Inputs from event handlers are generally translated into updates. For example, when several event handler components are represented by the same class, they may each cause a different update to occur. To do this, the class sends an input up the tree with an id indicating the action it represents. Another component will handle this input and send an appropriate update.

Task: Design flow of requests

A request is sent from an event handler whenever it requires a value held in, or computed by, a request handler. Requests can only move up the tree.

Working with the hand-drawing of all or part of the architecture, requests can be drawn in as labeled arrows connecting event handlers to request handlers. Another option is to list the names of requests made beside an event handler component. Some users do not add this level of detail to their hand-drawing.

Task: Design flow of updates

Updates are sent from event handlers whenever they need to set or change a value held by a request handler. Updates can only move up the tree.

Working from the hand drawing of the architecture or part of it, updates can be drawn as labeled arrows connecting event handlers to request handlers. Another option is to list the names of updates made beside an event handler component. Some users do not add this level of detail to their hand-drawing.

Task: Reuse a component

Components which have been defined previously in another application or which are part of a library of reusable components can be added to the architecture tree.

Care must be taken to make this component fit into the tree properly. Some modifications to surrounding components or to the reused component may be necessary.

Sub-Goal: Declare component classes

The object of this sub-goal is to produce the component class declarations which are used by the architecture file. These declarations may be part of the architecture file or they may be in separate files which are included in the architecture file when it is translated by Clock.

The declaration of component classes can be done in stages as the architecture structure is defined. Initial versions of these files containing only the class name for request handlers, and the class name and the subviews for event handlers can be used to run the CV tool and view the architecture structure.

Comments can be included in these files to make the system easier to follow as they became increasingly large.

Task: Declare an event handler

In this task, the event handler declarations are produced. By convention, each event handler class has its own file which has the same name as the event handler class, only beginning with a lower-case letter and ending with a .ehc extension.

The declaration includes the class name, and lists of subviews, inputs taken, requests made, and updates made for the event handler class.

Keywords used in this file are: *ehClass* (declares the name of the class), *subviews* (lists all subviews), *inputs* (lists input events received by this class), *requests* (lists requests sent by this class), and *updates* (lists updates sent by this class).

The pseudo-code for each event handler can be included in comment lines in the event handler declaration. This task has two purposes. First, it assists the task of verifying the completed architecture design. Second, it is a method of filling in the inputs, requests, and updates lists of the event handler class.

Task: Declare a request handler

In this task, the request handler declaration files are produced. By convention, each request handler class has its own file which has the same name as the request handler class, only beginning with a lower-case letter and ending with a .rhc extension.

The declaration includes the class name, and lists of requests and updates handled by this class.

Keywords in this file are: *rhClass* (declares the name of the class), *requests* (lists all requests taken) and *updates* (lists all updates taken).

Task: Verify syntax of a component class declaration

The CV tool can be used to verify the syntax of the component class declarations.

Task: Correct syntax of a component class declaration

To correct a syntax error in a component class declaration, the file is edited with any text editor. Declaration files cannot be edited from within the CV tool.

Task: Modify a component class declaration

Modifications to component class declarations are done with any text editor.

Sub-Goal: Create architecture file

An architecture file will be produced. This file includes all the necessary component class declarations and the textual definition of the architecture design.

Task: Include a required file

If a component class has been declared in a separate file, this file must be included in the architecture file using a *#include* directive. Declaration files which come from Clock libraries are included in the architecture file with a *#library* directive. The first two library files included should always be *StdUpdates.ct* and *StdRequests.ct*. These files include all the standard requests and updates used in the clock system.

The order in which component class declaration files are included in the architecture file is important. The best method is to include the request handler classes before the event handler classes. A commonly occurring problem is that a file will refer to a request or update which is defined in a later file. This condition generates a warning message indicating that a request or update has not been declared.

Task: Build architecture model in text

A textual definition of the architecture or part of the architecture, is created based on the hand drawing of its design. This file can be created with any text editor. The CV tool cannot be run until this file is created and syntactically correct.

The architecture file can be created in stages as the architecture model is being designed. An initial version of the architecture file can be written without request handlers. It is produced after the initial versions of the event handler class declaration files have been created. This initial architecture contains only the event handler classes and it shows the composition of the subviews used in the system.

Task: Modify architecture model textual description

Any text editor can be used to modify the architecture file. It cannot be modified from within the CV tool.

Task: Verify architecture file syntax

The syntax of the architecture file can be verified by running the CV tool. Error messages refer only to the translated version of the architecture file. This translated version includes all the component class declarations so that the error message does not indicate which file contains the error.

Task: Find an error in architecture file

When the CV tool is run, it stops at the first error it encounters and indicates the text surrounding the error. The error message refers only to the translated version of the architecture file (*.processedProgs/Architecture.ct)* which includes all the text from the declaration files as well as the Architecture file. There are several ways to find the error.

Sometimes the text provided in the error message is sufficient to find the error. If not, the line number of the error in the translated file is provided. By looking at this section of the translated file, more of the text which caused the error can be used to determine its location. Also, by looking at the lines above the error, the first line beginning with an ehclass or rhclass indicates the component class declaration which caused the error. The declaration file for this class usually contains the error.

Task: Correct a syntax error in architecture file

Once the error is located, it can usually be corrected by simple text editing. However, more serious errors may require more complex correction such as a change to the structure of the architecture design.

Sub-goal: Evaluate Architecture Design

Task: Verify position of a request handler

Request handlers should be placed as low as possible in the tree to prevent unnecessary redrawing of the screen. However, the request handler must be above any event handlers which send it requests or updates. The CV tool displays all the request and event handlers as coded in the architecture text file.

The picture created by the CV tool can be verified against the hand-drawing of the architecture design.

Task: Verify position of an event handler

The position of an event handler can be verified in several ways.

The position of an event handler can be checked against the UI design to ensure that all functional components have been included. The architecture text file can be checked against the hand-drawn architecture design to ensure that the text accurately corresponds to the design. The CV tool can be used to display the event handlers in the architecture text file.

Task: Verify component class declarations

Each request handler declaration is checked to ensure that it includes a declaration for every request and update it handles. Also, each request and update should have a request or update declaration and a type signature. This checking is not done by the CV tool. It is verified at compile time. Before compilation, the declarations must be checked manually, one at a time.

Task: Verify flow of requests

Requests are made whenever an event handler requires a value from a request handler. Requests must flow up the tree. The CV tool does not show the flow of requests. These must be verified using either the component class declarations, or the hand drawn architecture structure.

Task: Verify flow of inputs

Inputs flow from the user to event handlers or from one event handler to another. They must flow up the tree. The CV tool does not show the flow of inputs. These must be verified using either the component class declarations, or the hand-drawn architecture structure.

Task: Verify flow of updates

Updates flow between request handlers and event handlers. They must flow up the tree. The CV tool does not show the flow of updates. These must be verified using either the component class declarations, or the hand-drawn architecture structure.

Sub-Goal: Implementation

Implementation involves the actual coding of the components, compilation of the code, error checking and correcting, as well as a level of run-time checks done by the programmer to ensure that the program is working correctly. It also includes modifications made to the program after it has been implemented.

Since development is an incremental process, the application may be implemented in stages.

Code for components which have been reused from other applications or from a library of reusable components should be linked or copied to the current directory.

This code may require modifications to allow it to function properly within the current application.

Sub-Goal: Define event handlers

Task: Define an event function

Each input taken by an event handler is defined with an event function.

Task: Define the invariant function

The invariant function defines a condition which is always true. It is used to ensure that the view or state of a component remains consistent. Not all event handlers need an invariant function, but it must always be defined. The invariant function must be defined as noUpdate it if is not needed.

Task: Define the initially function

The initially function is called once when an event handler is being created. It has a single parameter which is the string passed to the subview when it is created. The initially function is used to set up initial values for an event handler.

Not all event handlers need an initially function, however it must always be defined. The initially function must be defined as noUpdate if it is not needed.

Task: Define the view function

In the initial implementation, the view functions are only done in rough form. Views are set up to stretch from the origin. They are initially set up above or beside each other. Once the code is running, the view function is modified and refined to correspond to the UI design.

Task: Declare each input taken as an update with a type signature

An input is treated as an update, so it needs an update declaration and a type signature.

The main problem with this task is that it is not always necessary. If an update is sent as an input to an event handler has already been declared elsewhere, then another declaration would cause an error. However, if it has not been declared, an error will occur.

Sub-Goal: Define a request handler

Task: Declare a type for the state

The state is the data being represented by the request handler.

Task: Define the initially function

The initially function defines the initial value for the request handler's state. It has no parameters.

Task: Define a request function

A request function has the same type as the request it handles except that it has an additional first parameter which is the request handler's state. For example, the request *myId* has type

myId :: String

whereas the function to handle the request has the type

myIdReq :: State -> String

The name of a request function is the same as the request it handles with an additional Req on the end. For example, the function to handle the *myId* request is called *myIdReq*.

Task: Define an update function

An update function has the same type as the update it handles except that it has a additional first parameter which is the request handler's state. For example, the update *setMyId* has type
setMyId :: String -> String,
whereas the update function has the type
setMyIdUpdt :: State -> String -> String.
The name of an update function is the same as the update it handles with an additional Updt on the end. For example, the function to handle the setMyId update is called setMyIdUpdt.

Task: Declare a request taken by request handler and its type.

Each request handled by a request handler must be declared with a request statement with a type signature.
This task causes a bit of confusion if it is unfamiliar. The type defined for a request in the request declaration is not the same as the type defined for the function which handles it. The type given in this declaration illustrates how the request will be called. The request function always has an additional first parameter which is the state of the request handler.
Also, the request and the request function have different names. For example, the request *myId* is handled by a function called *myIdReq*.

Task: Declare an update taken by request handler and its type

Each update handled by a request handler must be declared with an update statement with a type signature.
This task causes a bit of confusion if it is unfamiliar. The type defined for an update in the update declaration is not the same as the type defined for the function which handles it. The type given in this declaration illustrates how the update will be called. The update function always has an additional first parameter which is the state of the request handler.
Also, the update and the update function have different names. For example, the update *setMyId* is handled by a function called *setMyIdUpdt*.

Sub-Goal: Get application running

This goal is to get some version of the application compiled and running.

Task: Compile the Clock Code

Run *updatearch* once all the component classes have been defined.

Task: Check the syntax of the code

Syntax errors are detected during code compilation. However, each module can be looked over before compilation to ensure it is correct.

Task: Decipher error message

This task causes major problems because the error messages are not often incomprehensible. However, you can often tell if it is a syntax error or something more serious.

Task: Find error in code

Most of the error messages generated by Clock indicate indirectly which file contains the error. It indicates the gtml file which caused the error. The name of the gtml is similar enough to the name of the Clock file to see which Clock file contains the error. For example, if the Clock file is called *MyId*, then the gtml file will be *.processedprogs/gMyId.gtml*.

In some cases, the error occurs in a file apparently unrelated to any Clock files. If this happens, the best strategy is to look in the file indicated at the given line number and determine from the code written there which file contains the error.

The main problem is that most error messages are not helpful in determining anything but which Clock file contains the error. From this point, you have to look at the actual Clock file and find the error without guidance.

Task: Correct syntax errors by editing files

Specific details of this task depend upon the text editor being used to create and modify files.

Task: Modify a portion of the code

Sub-Goal: Debug the running application

Task: Fine-tune a view function

In this task, the view functions are fine-tuned according to the UI design. The running program can be used to determine which changes to make. The application must be recompiled after each adjustment. The UI design can be used to compute exactly which changes to make in the view functions.

Task: Detect run-time errors

Run-time errors can be detected only by running the program and trying out all the components and functions.

Task: Modify a portion of code

This task primarily involves text editing which depends upon the editor being used to create and edit files.

Task: Recompile code

Code is recompiled by issuing an *updatearch* command. This command recompiles only code which has been modified since the last compilation.

Sub-Goal: Modify the application

The biggest problem with modifying a Clock program is that the components are often inter-related in ways which cannot be seen by looking at the architecture design produced by the CV tool. When making modifications, care must be taken not to break the application.

Task: Define new components

When a new component is defined, it must be able to interact with the existing application without breaking it.

Task: Remove an existing component

To remove a component, care must be taken to modify any components connected to it.

Task: Modify existing components

Components can be modified by changing the functions which define them, or by changing their position in the architecture structure.

Sub-Goal: User Testing

This sub-goal is to evaluate the current version of the interactive application. Since the entire development is an iterative process, modifications may be made after the application has been tested and evaluated.

Appendix B: User Action Notation for *ClockWorks*

These UAN descriptions illustrate how the tasks identified in the task analysis (chapter 4) can be performed with *ClockWorks*. They correspond exactly to the task hierarchies illustrated in figures 4.2, 4.3 and 4.4. *ClockWorks* has eliminated many of the tasks identified. In these cases, a note has been added to the UAN description.

The following tasks have been added to the process of developing a Clock application because *ClockWorks* has changed the way in which Clock applications are developed. These tasks were added to *ClockWorks* as it was being implemented and therefore did not occur as part of the original task analysis:
• replace <component>
• duplicate <component>
• open <project>
• close <project>
• save <project>
• quit
• add <object> to library
Since these tasks did not appear as part of the original task analysis, their UAN description has not been included here.

Section 2.4 contains an explanation of User Action Notation.

Main Goal: Develop and Interactive Application

User Action	Notes
user needs analysis	not supported by environment
(user interface design	not supported by environment
task oriented specification)+	not supported by environment
(architecture design	expanded below
÷ implementation)+	expanded below
user testing	not supported by environment

Sub-Goal: Architecture Design

User Action	Notes
(design tree structure	(addEH | addRH | addUpdt | addInpt | addReq)*
÷ declare component classes	this is done automatically by the new environment (addEH | addRH | addUpdt | addInpt | addReq)*
÷ create architecture file	this is done automatically by the new environment
÷ evaluate architecture design)	many tasks can be used to achieve this goal

Sub-Goal: Implementation

User Action	Notes
((define an event handler)+	open <ehObject> <edit> edit <ehObject>
÷ (define a request handler)*)	open <rhObject> <edit> edit <ehObject>
(get application running	many tasks can be used to achieve this goal
÷ debug running application	many tasks can be used to achieve this goal
÷ modify code)+	many tasks can be used to achieve this goal

Sub-Goal: Design tree structure

User Action	Notes
((add an event handler)+	addEH
÷ (add a request handler)*	addRH
÷ design flow of inputs	addInpt
÷ design flow of requests	addReq
÷ design flow of updates)*	addUpdt

Sub-Goal: Declare component classes

User Action	Notes
declare an event handler	done automatically by environment when user adds an event handler and gives it a name addEH
declare a request handler	done automatically by environment when user adds a request handler and gives it a name addRH
verify syntax of component class declarations	done automatically by environment
correct syntax of component class declarations	since declarations are created by the environment, there should be no need to correct the syntax
modify component class declarations	user does not directly modify the component class declaration files, as they are automatically generated by the environment (addUpdt | addReq | addInpt | addEH)*

Sub-Goal: Create the architecture file

User Action	Notes
((include required files	done by environment automatically according to class names chosen by developer
| include component class declarations)+	done by environment automatically according to class names chosen by developer
(build architecture text)+	textual version of architecture is hidden from the user and created automatically
(modify architecture)*	user is not permitted to modify text directly, architecture is modified by direct manipulation
(verify architecture file syntax)*	since environment creates the architecture file, the syntax is error free
(find an error in architecture file)*	syntax is generated correctly by environment
(correct an error in architecture file)*)+	architecture file is generated automatically and will not contain errors

Sub-Goal: Evaluate architecture design

User Action	Notes
verify position of request handlers	by looking at architecture the in various modes provided, the user will be able to verify the position of request handlers
verify position of event handlers	by looking at architecture in the various modes provided, the user will be able to verify the position of event handlers
verify component class declarations	component class declarations will be hidden from the user, modifications to any part of architecture design will be done by direct manipulation
verify flow of requests	various display modes can be used to verify flow of requests
verify flow of inputs	various display modes can be used to verify flow of inputs
verify flow of updates	various display modes can be used to verify flow of updates

Sub-Goal: Define an event handler

User Action	Notes
event function	no change
invariant function	no change
initially function	no change
view function	no change
define type for inputs	defineType<inptObject>

Sub-Goal: Define a request handler

User Action	Notes
Define type for request handler state	no change
initially function	no change
request functions	no change
update functions	no change
define type for requests	defineType<reqObject>
define type for updates	defineType<updtObject>

Sub-Goal: Get application running

User Action	Notes
compile	choose <special><compile>
Check syntax	no change
decipher error message	no change
find error	no change
correct syntax	no change
modify code	open <component><edit> edit <component>

Sub-Goal: debug running application

User Action	Notes
fine tune view function	no change
detect run-time errors	no change
modify code	open <component> <edit> edit <component>
compile code	choose <compile> <special>

Sub-Goal: Modify application

User Action	Notes
(define new object |	(addEH | addRH | addInpt | addReq | addUpdt)*
remove a component |	remove <object>
modify component)*	(addInpt | addReq | addUpdt | edit<component>)*

Task: AddEH

User Action	Interface Feedback	Connection to Computation
select <ehObject>	<ehObject> !	add <ehObject> to selected list
choose <class> <add subview>	event handler dialog appears	see figure 6.4
name ehClass <new eh>	as in task	as in task

Task: AddRH

User Action	Interface Feedback	Connection to Computation
select <ehObject>	<ehObject> !	add <ehObject> to selected list
choose <Component> <add request handler>	request handler dialog box appears	see figure 6.5
name rhClass <new rh>	as in task	as in task

Task: AddInpt

User Action	Interface Feedback	Connection to Computation
select <ehObject>	<ehObject>!	add <ehObject> to selected list
choose <class> <add input>	input, request, update declaration dialog box appears	see figure 6.6
inptName <inptObject>	as in task	as in task
defineType <inptObject>	as in task	as in task

Task: AddReq

User Action	Interface Feedback	Connection to Computation
(select <ehObject>	<ehObject>!	add <ehObject> to selected list
choose <class> <add request>	input, request, update declaration dialog box appears	see figure 6.6
reqName <reqObject>	as in task	as in task
defineType <reqObject)	as in task	as in task
| (select <rhObject>	<rhObject>!	add <rhObject> to selected list
choose <class> <add request>	input, request, update declaration dialog box appears	see figure 6.6
reqName <reqObject>	as in task	as in task
defineType <reqObject>)	as in task	as in task

Task: AddUpdt

User Action	Interface Feedback	Connection to Computation
(select <ehObject>	<ehObject>!	add <ehObject> to selected list
choose <class> <Add Update>	input, request, update declaration dialog box appears	see figure 6.6
updtName <updtObject>	as in task	as in task
defineType <updtObject>)	as in task	as in task
| (select <rhObject>	<rhObject>!	add <rhObject> to selected list
choose <class> <add update>	input, request, update declaration dialog box appears	see figure 6.6
updtName <updtObject>	as in task	as in task
defineType <updtObject>	as in task	as in task

Task: Name ehClass

User Action	Interface Feedback	Connection to Computation
if <ehDialog> not active:	MainScreen is displayed (currently)	see figure 6.3
select <ehObject>	<ehObject>!	add <ehObject> to selected list
choose <class> <rename>	event handler dialog box appears	see figure 6.4
end if	eh name dialog is displayed (currently)	see figure 6.4
(~<name> ts |	<name> ! <name> appears in <editBox>	
(~<editBox> ts)	cursor appears in <editBox>	
type <name>))	echo characters in <editBox>	
~ <OK> ts	<OK>!-! rh name dialog goes away MainScreen is displayed	see figure 6.3

Task: Name rhClass

User Action	Interface Feedback	Connection to Computation
if rh name Dialog not active:	MainScreen is displayed (currently)	see figure 6.3
select <rhObject>	<rhObject>!	add <rhObject> to selected list
choose <class> <rename>	request handler dialog appears	see figure 6.5
if rh Name dialog is active	request handler dialog is displayed (currently)	see figure 6.5
(~<name> ts |	<name> ! <name> appears in <editBox>	
(~<editBox> ts)	cursor appears in <editBox>	
type <name>))	echo characters in <editBox>	
~ <OK> ts	<OK>!-! request handler name dialog goes away MainScreen is displayed	see figure 6.3

Task: DefineEH <ehObject>

User Action	Interface Feedback	Connection to Computation
setLevel <ehObject> <edit>	<ehObject> code is displayed in text editor	
edit <ehObject>	specific to chosen text editor	

Task: DefineRH <rhObject>

User Action	Interface Feedback	Connection to Computation
setLevel <rhObject> <edit>	<rhObject> code is displayed in text editor	
edit <rhObject>	specific to chosen text editor	

Task: Show SrcDest <iruObject>

User Action	Interface Feedback	Connection to Computation
select <iruObject>	src/dest !-!	

Task: Open <object>

User Action	Interface Feedback	Connection to Computation
(~<object> tsts) |	open object to next level if already open to highest level, do nothing	see figure 5.8 for an explanation of levels
(select <object>	<object> !	add <object> to selected list
choose <edit> <open>)	open object to next level if already open to highest level, beep	see figure 5.8 for an explanation of levels

Task: Close <object>

User Action	Interface Feedback	Connection to Computation
select <object>	<object> !	add <object> to selected list
choose <edit> <close>	object closes to next level. if already at lowest level, do nothing	see figure 5.8 for an explanation of levels

Task: select <object>

User Action	Interface Feedback	Connection to Computation
if not selected <object>:	<object> -! (currently)	<object> is not in selected list
~ <object> ts	object !	add <object> to selected list

Task: unSelect <object>

User Action	Interface Feedback	Connection to Computation
if selected <object>:	<object> ! (currently)	<object> is part of selected list
(~ <object> ts |	<object> -!	remove <object> from selected list
~ <blank space> ts)	<object>-! and <all selected objects> -!	remove all selected objects from selected list

Task: defineType <iruObject>

User Action	Interface Feedback	Connection to Computation
if input, request, update declaration dialog not active:	MainScreen is displayed	see figure 6.3
select <iruObject>	<iruObject>! <src or dest of iruObject>!	
choose <class> <IRU declaration>	input, request, update declaration dialog box appears	see figure 6.6
if input, request, update declaration dialog active:	input, request, update declaration dialog is displayed	see figure 6.6
~ <type editfield> ts	cursor appears in type edifield	
type <type signature>	echo typing	
~ <OK>	<OK> !-! name-type dialog goes away MainScreen appears	see figure 6.3

Task: selectGroup

User Action	Interface Feedback	Connection to Computation
<shift> t		
((select <object>)+	<object> !	add <object> to selected list
(unSelect <object>)*)	<object> -!	remove <object> from selected list
<shift> s		

Task: unSelect group

User Action	Interface Feedback	Connection to Computation

~ <anywhere> ts	if clicked on blank space:
	all object unselected
	if clicked on <new object>:
	all objects unselected
	<new object> selected
		update selected list to reflect changes

Task: Group

User Action	Interface Feedback	Connection to Computation
select<groupObject>	as in task	as in task
choose <group><group>	selection becomes a single group object	update tree structure to reflect changes

Task: UnGroup <groupObject>

User Action	Interface Feedback	Connection to Computation
select <groupObject>	<groupObject> !	add <groupObject> to selected list
choose <group><ungroup>	group object becomes original portion of architecture	update tree structure to reflect changes

Task: choose <menu> <item>

User Action	Interface Feedback	Connection to Computation
~ <menu> t	<menu> pops down	
~ <item>	<item> !	
^	<item> -! <menu> pops up	perform task associated with this menu item

Task: inptName <inptObject>

User Action	Interface Feedback	Connection to Computation
if iru declaration dialog not active:	MainScreen is displayed	see figure 6.3
select <inptObject>	<inptObject>! <src or dest of inptObject>!	add <inptObject> to selected list
choose <class> <rename>	iru declaration dialog appears	see figure 6.6
if iru dialog is active:	iru dialog is displayed (currently)	see figure 6.6
(~<name> ts |	name appears in name box	
(~ <name edit> ts	cursor appears in <name edit>	
type <name>))	echo characters	
~ <OK> ts	<OK> !-! iru dialog goes away MainScreen activated	see figure 6.3

Task: reqName <reqObject>

User Action	Interface Feedback	Connection to Computation
if iru declaration dialog not active:	MainScreen is displayed	see figure 6.3
select <reqObject>	<reqObject> ! <src or dest of reqObject)!	add <reqObject> to selected list
choose <class> <rename>	input, request, update dialog appears	see figure 6.6
if iru declaration dialog is active:	input, request, update dialog is displayed (currently)	see figure 6.6
(~<name> ts |	name appears in name box	
(~ <name edit> ts	cursor appears in <name edit>	
type <name>))	echo characters	
~ <OK> ts	<OK> !-! iru dialog goes away MainScreen activated	see figure 6.3

Task: updtName <updtObject>

User Action	Interface Feedback	Connection to Computation
if iru dialog not active:	MainScreen is displayed	see figure 6.3
select <updtObject>	<updtObject> ! <src or dest of updtObject> !	add <updtObject> to selected list
choose <class> <rename>	input, request, update dialog appears	see figure 6.6
if iru dialog is active:	input, request, update dialog is displayed (currently)	see figure 6.6
(~<name> ts |	name appears in name box	
(~ <name edit> ts	cursor appears in <name edit>	
type <name>))	echo characters	
~ <OK> ts	<OK> !-! iru dialog goes away MainScreen is activated	see figure 6.3

Task: ehName

User Action	Interface Feedback	Connection to Computation
if event handler dialog not active:	MainScreen is displayed	see figure 6.3
select <ehObject>	<ehObject> !	add <ehObject> to selected list
choose <class> <rename>	event handler dialog appears	see figure 6.4
if event handler dialog is active:	event handler dialog displayed	see figure 6.4
~ < nameEdit> ts	cursor appears in name edit	
type <name>	echo characters	
~<OK> ts	<OK>!-! event handler dialog goes away display MainScreen	see figure 6.3

Task: SetLevel <object> <level>

User Action	Interface Feedback	Connection to Computation
(open <object> |	<object> will be displayed in the next highest level	see figure 5.8 for an explanation of levels
close <object>)*	<object> will be displayed at the next lowest level	see figure 5.8 for an explanation of levels
until <level> reached		

Task: Remove <object>

User Action	Interface Feedback	Connection to Computation
select <object>	<object> !	add <object> to selected list
choose <edit> <remove>	object deleted from tree and tree repaired to fill in the gap	restructure tree to fill in gap