

ClockWorks: Visual Programming of Component-Based Software Architectures

T.C. Nicholas Graham

Catherine A. Morton

Tore Urnes

Department of Computer Science
York University
4700 Keele St., North York
Canada M3J 1P3
graham@cs.yorku.ca

Correspondence should be directed to:

T.C. Nicholas Graham

Department of Computer Science
York University
4700 Keele St., North York
Canada M3J 1P3
graham@cs.yorku.ca

Abstract

ClockWorks is a programming environment supporting the visual programming of object-oriented software architectures. In developing ClockWorks, we used user interface evaluation techniques, including heuristic evaluation, cognitive walkthrough and user evaluation. The development of ClockWorks was based on a task analysis of ClockWorks programmers. This task analysis revealed that programmers work incrementally. Incremental development implies the need for good support for information filtering and for easy refinement and restructuring of programs. ClockWorks has been implemented, and runs on Sun workstations. All examples shown in this paper were programmed with ClockWorks.

Keywords: Visual Programming, User Interface Architectures.

1 Introduction

In 1991, we began a project to develop tool support for programming component-based software architectures, with the particular application domain of software architectures for graphical user interfaces. Four years later, we have completed the third version of *ClockWorks*, a visual programming environment for software architectures. This version is the first one which programmers actually prefer to use over the old textual representation of software architectures. This paper details what we have learned, through our experimentation and many missteps, about the human factors of visual programming languages. ClockWorks is a tool for programming software architectures, and so is somewhat simpler than fully general visual programming languages. Nevertheless, we believe that our experiences are also of interest to people developing more complex languages than ours.

Our experiences can be summarized in three main points:

- *It's easy to create information overload:* It is easy to display lots of useful and interesting information for the programmer. What is hard is determining what *not* to display, so that programmers do not become overwhelmed with detail. In early versions of *ClockWorks*, information overload was the primary reason programmers went back to a textual notation. Support must be built right into the language to help make it easy for the programmer to filter information. Relatedly, in visual languages, “large” isn’t very large – scalability problems due to information overload or too little screen real estate crop up in relatively small visual programs.
- *Interacting with visual programs is hard:* Developing a visual notation for some programming domain is relatively easy. Effectively displaying visual programs on a fixed-sized display is harder. Providing effective editing support for visual programs is very hard. VPL designers must think of their notations not as being static pictures, but as pictorial views of a model that programmers will wish to view in many different ways, and that will evolve significantly over time. Unlike textual languages, where editing and viewing support are not part of the language design, browsing and editing support are an integral part of visual languages.

- *VPL design is UI design:* Our group had significant experience in both programming language and user interface design. Due to the very interactive nature of visual languages, we found VPL design to be closer to the second of these activities. Techniques from user interface design were useful in evolving the design of ClockWorks to a usable state, particularly the techniques of task analysis, heuristic evaluation, cognitive walkthrough, and usability testing. Designers of visual languages should build these steps into the language design process, rather than assuming that static pictures can be easily mapped into a successful programming environment.

The paper is structured as follows. First, we briefly introduce the Clock programming language and the ClockWorks programming environment. We then detail the process used to develop ClockWorks, leading to the results of our task analysis of Clock programmers. The remainder of the paper shows how ClockWorks was designed to support these tasks.

2 Overview of Clock and ClockWorks

The *Clock* language [1] is intended to support the prototyping of graphical user interfaces, concentrating particularly on the development of multi-user interfaces and groupware. Clock consists of a visual architecture language, used to specify the class, visibility and compositional structure of Clock programs [2]. Clock architectures are made up of components, which are themselves programmed textually, based on the functional language Haskell [3]. This paper concentrates on the visual programming of Clock architectures.

Figure 1 shows an example program written in Clock, a terminal reservation system as might be used in the York University undergraduate computing lab. Students are permitted up to three one-hour reservations per week. To make a reservation, the student selects a terminal, selects a duration, and clicks on “*Reserve*”. Terminals are colour-coded to indicate how long they are free, ranging from grey (not free at all), to dark green (free for three hours or more.)

Figure 2 shows how the architecture for this program is represented in *ClockWorks*. ClockWorks is a visual programming environment for Clock, allowing architectures to be built, edited, and executed by direct manipulation. Menus provide access to facilities for editing

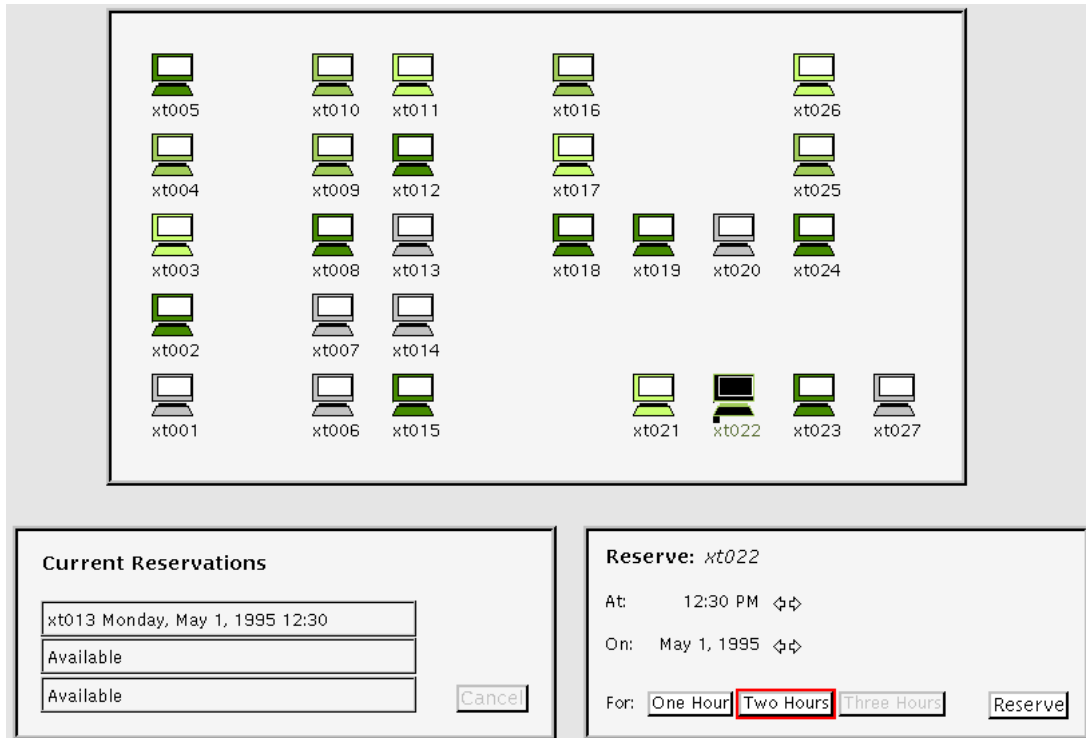


Figure 1: The display of a terminal reservation system written in Clock. Colour coding is used to indicate how long a terminal is free – grey terminals are not free at all; light green are free for one hour; dark green are free for three hours or more. Users may reserve up to three one-hour slots per week.

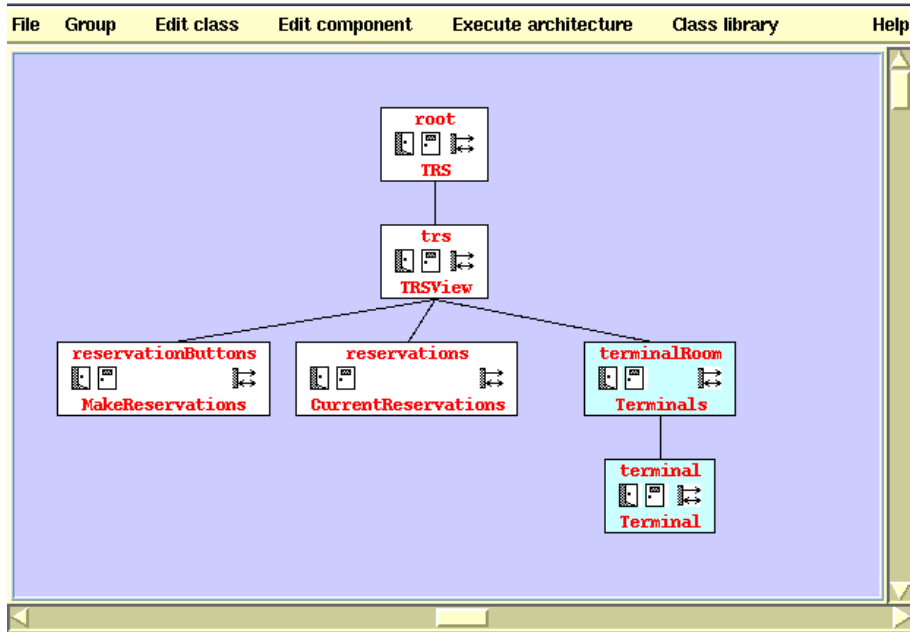


Figure 2: The terminal reservation system from figure 1 as displayed in the ClockWorks visual programming environment.

architectures and executing them, and to libraries of predefined component classes.

Clock architectures are structured as trees of components, representing the hierarchical structure of the user interface being developed. In figure 2, the *TRSView* component is composed of a set of *reservationButtons*, a panel showing the current set of *reservations*, and a *terminalRoom* showing the map of terminals. Whenever a new architecture component is defined, an associated class is automatically defined. For example, the component *trs* is defined to be of class *TRSView*. Any modification to *trs* is therefore considered also to be a modification to the *TRSView* class.

Icons on the components allow easy elision and expansion of detail: the open and close-door icons specify that more or less detail should be shown respectively. The arrows icon is used to toggle the display of the component’s *interface*, showing which methods the component implements and uses.

Figure 3 shows the implementation of the *SimpleButton* class used to implement the “*Cancel*” and “*Reserve*” buttons in the terminal reservation system. View (a) shows the interface

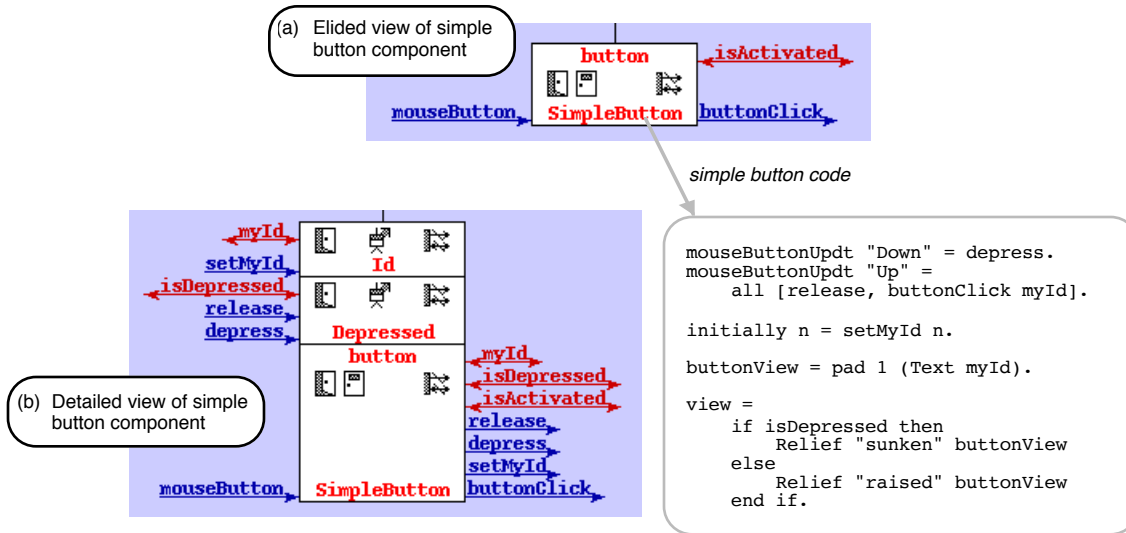


Figure 3: The interfaces of components can be displayed by clicking on the arrows icon. Components can be “opened up” by clicking the open door icon on the component. The interfaces of grouped components are automatically elided to show only non-local updates and requests.

of the button: it is capable of handling “*mouseButton*” inputs (received when the button is clicked), and issues the “*buttonClick*” update when the button is released. The button also uses the request “*isActivated*” to determine whether the button is currently enabled for use. (For example, in figure 1, the “*Reserve*” button is activated, while the “*Cancel*” button is not.) Requests and updates are messages that are sent up the tree to the first component capable of handling them. Therefore, any component is capable of accessing any data represented above it in the tree.

View (b) of figure 3 shows a detailed view of the button component. The component uses two *request handlers* to represent its state: the *Id* request handler maintains the text appearing on the button, and the *Depressed* request handler records whether the button is currently depressed or released. In the detailed view, we see the updates and requests implemented by *Id* and *Depressed*, and which of them are used in *SimpleButton*. In the elided version of view (a), these *local* requests and updates are not shown.

For completeness, figure 3 also shows the textual code that is used to implement the button component itself.

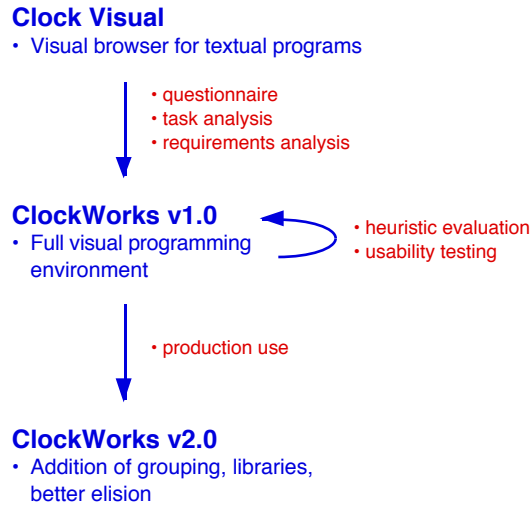


Figure 4: The process used to develop the ClockWorks visual language. This process is primarily based on techniques from human-factors research.

3 Process

Before proceeding to the detailed design of ClockWorks, we shall describe the process used in developing ClockWorks. This approach is based heavily on techniques used to design user interfaces; many of these techniques are described in standard texts [4, 5]. Our approach was also heavily influenced by the work in user interface evaluation of Jeffries et al. [6], and the work in user interface specification of Hartson et al. [7].

Figure 4 shows the process followed, leading us to the current design of ClockWorks. From the first origins of the language in 1991, Clock’s architecture language has had a visual syntax. The first version of the language, however, obliged programmers to enter architecture descriptions in a textual language. The *Clock Visual* tool was available to programmers to visually browse architectures that had been entered in this textual form. The main users of this tool were graduate students working with Clock, and 25 students in an upper year course on human-computer interaction.

To evaluate the effectiveness of the Clock Visual tool, a questionnaire was circulated among all Clock users. The results of the questionnaire indicated clearly that the tool was not being used: programmers preferred to work with the textual representation of architec-

tures rather than using the visual tool. Programmers continued to draw diagrams of their architectures on paper, indicating that the visual syntax was how they naturally thought of architectures. Clearly, simply providing a visual representation of Clock architectures was not sufficiently useful. The questionnaire illustrated that over all, programmers found programming in Clock to be very difficult, but did not provide a clear understanding of why this was so. One issue that was clear was that Clock programmers suffered a form of information overload, where the complex communication among components became too hard to understand.

To better understand why the Clock Visual tool was not successful, we performed a *task analysis* of Clock programmers. The goal of the task analysis was to determine the way in which Clock programmers develop programs, and to break down this development process into its constituent tasks. The task analysis was largely done through interviews with programmers, and examination of the hand-drawn diagrams that programmers produced. The full details of the task analysis can be found in [8]. From the task analysis, a set of requirements for the new *ClockWorks* tool was developed.

ClockWorks was designed, and tested for usability through heuristic evaluation and by using the User Action Notation [7] to show that each of the programmer's tasks could all be reasonably performed. Finally, the new system was implemented, and evaluated with user testing, following the method of Gomoll's ten steps [9].

The resulting tool, ClockWorks 1.0, was released into production use within our research group. During this use, it became evident that the programmer's tasks had changed substantially from those identified in the earlier study. Largely, this came about because the new tool so simplified programming in Clock that many earlier tasks were no longer necessary. New tasks also were related to how to deal with large programs, because the new system allowed programmers to work with much larger programs than was feasible earlier, since problems with information overload had been greatly reduced.

Based on the experiences with using ClockWorks, a completely new version 2.0 of the tool was developed, building in far better support for large Clock programs.

From our experience with the various versions of the ClockWorks tool, it is clear that

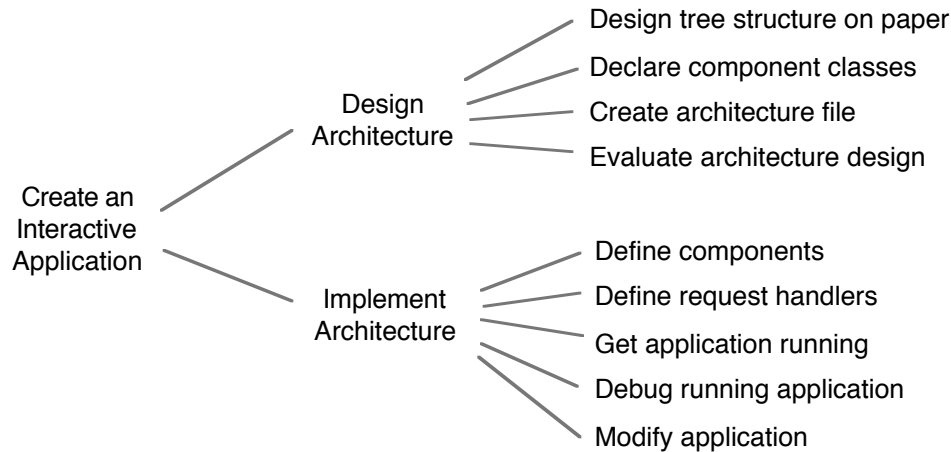


Figure 5: The original task analysis of Clock programmers preceding the design of ClockWorks v1.0.

the success of visual languages is highly dependent upon the programming environments supporting their use. Even in a relatively simple language such as Clock, early versions of the language were not used, because the tools failed to properly support the programming tasks the programmers wished to perform. It is only through properly understanding the programmers' tasks and through many iterations of usability testing that tools for visual languages can become good enough for production use.

The next section details what we found out through the task analysis of Clock programmers, and how we chose to support these tasks in the ClockWorks environment.

3.1 Task Analysis of Visual Programmers

In order to determine what support programmers require in a visual programming environment, a *task analysis* can be performed. A task analysis consists of finding out how programmers produce programs. The design of a visual programming environment then is based on trying to optimize the programming process by eliminating unnecessary tasks, and by providing good support for necessary tasks. As will be shown in the following sections, task analysis was useful in the design of ClockWorks in that it helped explain why the original Clock Visual tool was not successful, and helped to track the changes in the way

programmers used the Clock language as the environment evolved.

As indicated by the process of figure 4, after a questionnaire had demonstrated that programmers did not find the Clock Visual tool to be useful, we performed an analysis of the tasks involved in creating Clock programs. The result of this analysis is shown in figure 5. The analysis was based on interviews with Clock programmers, and examination of design documents produced by programmers. The task analysis showed that creating an application consisted of two sharply separated steps: first designing an architecture, and then implementing it. The task analysis helped to reveal that once a programmer entered an architecture into the system, he/she rapidly ceased to be able to understand it, and hence was unable to effectively modify it. Programmers therefore felt a need to completely design their system on paper, carefully verify that the architecture was complete and correct, and only then enter it into the system. Programmers were not using the Clock Visual tool because it did not provide sufficient support for understanding programs. This lack of support forced programmers to work instead with hand-drawn representations of their architectures.

In the interviews associated with the task analysis, programmers cited the following reasons why the Clock Visual tool did not adequately support program understanding: (a) the tool did not provide complete access to all information in the architecture, and (b) since the tool provided visualization only, programmers still had to work with the textual representation to make changes to their architectures. The development of ClockWorks 1.0 was largely motivated by the desire to solve these problems by providing a full visual programming environment for Clock software architectures. The initial task analysis of programming in Clock therefore helped to explain why programmers did not use the visual tool we had provided. The analysis also helped to determine the requirements of ClockWorks v1.0, the system intended to replace Clock Visual.

3.1.1 Evolution in Task Analysis

The designers of a language cannot predict how the language will be used in practice [10]. Once they are made available to users, languages must evolve in response to the problems that the users discover. Similarly, a task analysis must evolve to reflect the changes in

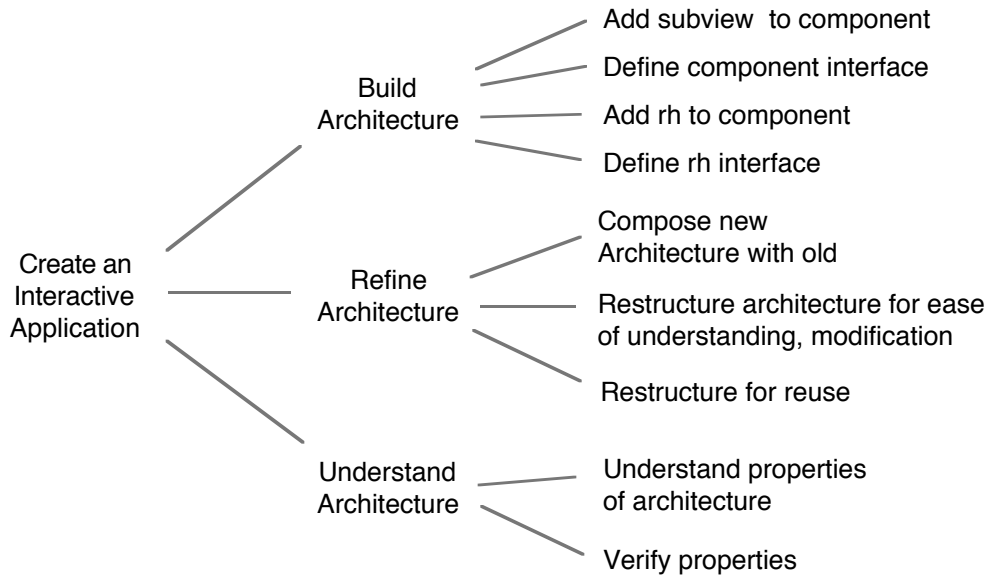


Figure 6: The revised task analysis of Clock programmers preceding the design of ClockWorks v2.0.

the way programmers use the visual programming environment. This evolution helps to explain how problems arising in a new environment may be the result of evolving use by programmers, and helps the designer document the intended use of a revised environment. This section shows how task analysis helps track evolution in the use of a language and in requirements for environment support.

ClockWorks v1.0 was developed based on the task analysis of figure 5 [8]. Once this environment was complete and being used by programmers, we saw an interesting evolution in the way in which programs were developed. ClockWorks v1.0 provided vastly improved support for program understanding. This finally allowed programmers to work in a truly incremental style. We began to observe that programmers were now developing architectures in small parts, getting those parts working, and extending and composing them into more complex architectures. At this point, programmers began to encounter the following difficulties:

- Architectures were getting so large that they once again became difficult to understand;

- Although the environment in principle supported the refinement of architectures, the facilities for refinement were sometimes clumsy and unintuitive;
- As architectures became larger, programmers wanted improved facilities for reusing parts of architectures, and for restructuring architectures to make them more easily reusable.

In response to these difficulties, the updated task analysis of figure 6 was developed. This task analysis showed that the initial success of ClockWorks v1.0 had given way to further difficulties. The task analysis shows that developing an interactive system consists of building, refining and understanding an architecture. These activities are carried out repeatedly and incrementally, until the application is finally developed. In the updated task analysis, we switched from describing how the language was currently being used, to describing how programmers wished they could program. The development of ClockWorks v2.0 was motivated by the desire to support the tasks identified in figure 6.

3.1.2 The Contribution of Task Analysis

Task analysis contributes to the design of visual programming environments in the following ways. The very fact of performing a task analysis acknowledges that in designing a visual programming environment, not just the static notation of the language is important, but also how the language is used. The task analysis helps find problems with how a visual programming language is being used, and helps document how programmers wish they could program. The evolving task analysis also helps track the changing use of a language, allowing designers to anticipate how the environment should evolve.

The one drawback of task analysis is that it describes either how the language is currently used, or how the designer intends that it be used. In the former case, there is a danger of designing an environment that simply implements the identified tasks, rather than seeing how these tasks could be simplified or eliminated. In the latter case, the designer may incorrectly anticipate how the language will be used. The task analysis of figure 5 led us to an environment which ultimately did not anticipate the extent to which programmers

would rely on understanding of large architectures and sophisticated refinement and reuse of architectures. Task analysis must therefore be seen as part of an iterative process for the design of visual programming languages.

3.2 Evaluation of the Design of a Visual Programming Environment

In developing the two versions of ClockWorks, as shown in figure 4, we employed a series of techniques for evaluation of user interfaces. This section details our use of *heuristic evaluation* and *task-oriented specification* to evaluate the design of ClockWorks v1.0 prior to its implementation. Section 3.3 shows how *user observation* was applied to evaluate the first prototype of ClockWorks v1.0.

3.2.1 Heuristic Evaluation

The idea behind *heuristic evaluation* [6] is that a user interface expert can analyze a user interface based on his/her experience, and detect flaws in the user interface design. The main advantage of heuristic evaluation is that errors can be detected before the design is implemented, saving expensive reimplementation. Nielsen [5] cites some basic guidelines followed in heuristic evaluation; these include verifying that the user interface is consistent (that is, similar tasks are carried out via similar mechanisms); that memorization is minimized (ie, that the user can work out how to proceed without having to memorize a large command set, or memorize information to be carried from one screen to another), and that frequently performed operations are optimized. This form of evaluation allowed us to fix numerous minor design errors in ClockWorks.

3.2.2 Task-Oriented Specification

A very useful technique for evaluating a design before its implementation is *task-oriented specification*. The idea behind task-oriented specification is to demonstrate that each of the tasks identified in the task analysis can be carried out with the design. We carried out this process using the *User Action Notation* [7]; since the UAN is a precise notation, the

Task: Develop an Interactive System
USER ACTIONS
Build architecture \Leftrightarrow Refine Architecture \Leftrightarrow Understand Architecture

(a) Developing an interactive system is split into building, refining and understanding an architecture. These tasks can be interleaved.

Task: Move RH (r, e1, e2)
USER ACTIONS
delete RH (r, e1)
add RH (classOf (r), e2)

(b) To move a request handler r from component e1 to e2, r must first be deleted from e1, and then a new request handler of the same class as r must be instantiated in e2.

Figure 7: Simple examples of UAN task-oriented specifications.


Task: Move RH (r, e1, e2)	
USER ACTIONS	INTERFACE FEEDBACK
\sim [ Mv	r!
\sim [x,y]	r moves to (x,y)
\sim [e2]	e2!
M^	e2-! r-! Tree redrawn, r now belonging to e2



Figure 8: The *Move RH* task from figure 7(b), updated for ClockWorks v2.0.

resulting document also serves as a specification of the visual programming environment suitable for implementation by a programmer. As shown in figure 4, we used the UAN to perform a task-oriented specification of ClockWorks v1.0 prior to its implementation.

UAN specifications differ from traditional specifications in that they are written from the point of view of a user of the system. The UAN specification shows how a user of the system could carry out each of the tasks in the task analysis. For example, in figure 7(a), it is shown how users build an interactive system, as following the task analysis of figure 6. The task of developing an interactive system is split into the three tasks of building an architecture, refining the architecture, and understanding the architecture. These tasks may be carried out in any order, or more typically *interleaved* (as specified by the “ \Leftrightarrow ” symbol), meaning that all three can be carried out at once. This notation succinctly captures

the idea that ClockWorks programmers may work incrementally, building part of their architecture, examining its properties, and continuing to refine it. Temporal relationships such as interleaving are hard to express in the task analysis itself. The UAN notation exposes such temporal relationships, making it easier to verify that a given design supports not only the necessary tasks, but also the desired interactions between tasks.

The UAN is also highly useful in identifying which tasks are poorly supported by the environment. The UAN specification of ClockWorks v1.0 was based on the task analysis of figure 5. This specification showed, for example, that ClockWorks v1.0 did not provide good support for restructuring architectures. While all restructuring operations were in fact possible, some were clumsy, and many were not obvious to naive users of the system. For example, figure 7(b) shows how the task of moving a request handler r from a component e_1 to another component e_2 was performed. (This task forms part of the *Refine Architecture* task.) In ClockWorks v1.0, to move a request handler, it is first deleted from the original component, and then a new instance of the same request handler class is instantiated in the new component. This specification makes use of two subtasks *delete RH* and *add RH*, not defined here. Similarly, moving a subtree consisting of several components required a series of delete and reinstantiate operations. Restructuring based on first deleting components and then reinstantiating them is clumsy and unintuitive.

Figure 8 shows how the *Move RH* task is implemented in ClockWorks v2.0. The improved mechanism for moving request handlers is based on the introduction of a new “” icon. As shown in the UAN, request handlers can be repositioned by first clicking down on the “” icon, and dragging the request handler to its new location. The UAN shows precisely how this is performed: the user moves the mouse over the icon, clicks down (shown by the “Mv”), at which point the request handler is highlighted (“r!”). As the mouse is moved, the request handler is dragged to the same position. When the mouse is moved over the target component and released, the highlighting is removed from the request handler, and the tree is redrawn with the request handler in the new position. The UAN specification clearly shows that this is a far more intuitive and direct mechanism for restructuring than the delete-and-reinstantiate mechanism from figure 7(b). Using the UAN, it is possible to examine the support that an environment provides for complex operations before the

environment has been implemented. This allows design errors to be found and addressed before expensive implementation has been performed.

In summary, the primary benefit of the UAN approach to task-oriented specification is that it is possible to test how well the design of a visual programming environment supports the tasks we have identified as important, before the environment is implemented. The UAN notation is simple, terse and expressive, allowing clear and precise documentation of how tasks are to be carried out. The UAN exposes the temporal relationships among tasks, introducing more information than is practical in the task analysis itself.

The major disadvantage of a task-oriented specification is that it is only as good as the task analysis that underlies it. If the task analysis does not adequately reflect the tasks that programmers really need to perform, then failure to provide support for these tasks will not be detected. In our UAN specification of ClockWorks v1.0, the task of modifying an architecture was not given sufficient prominence, allowing us to miss the difficulty of refining complex architectures. Therefore, it must be seen that task-oriented specification does not eliminate the iterative nature of the visual programming environment design process, but rather helps increase the amount of progress that can be made in each iteration.

3.3 Gaining Feedback from Users

Once a prototype of a visual programming environment is available, it is important to test it with real users. User testing has the two major benefits of identifying small but serious errors that a designer could not have anticipated, and of showing how real users use the environment in ways which the programmer did not anticipate. In the development of ClockWorks, experience with users played an important role in both of the iterations of development that have occurred so far. This experience has led us to see that not all forms of user feedback are useful: direct user observation using Gomoll's ten steps [9] was very useful; questionnaires and user surveys were of limited use. In this section, we give an overview of each of these three methods.

3.3.1 Questionnaires

As indicated in figure 4, before undertaking the development of ClockWorks, we circulated a questionnaire among approximately twenty Clock programmers. The purpose of the questionnaire was to ascertain whether the programmers were using the existing Clock Visual tool at all, and what problems they were experiencing with the tool. Our hope was that their detailed responses would give us the first step in designing the new ClockWorks environment. The results of the questionnaire served primarily to show that the ClockVisual tool was not widely used. The surveyed programmers were, however, unable to clearly articulate what the problems were that they faced with the tool.

3.3.2 User Survey

To find information about what does and does not work with a user interface once it has been implemented, feedback from users is required. Our first attempt to evaluate ClockWorks v1.0 was based on asking two existing Clock programmers to try out the new environment, and to report their findings to us. These users were experts in the Clock language, and were interested in seeing better tools made available. The feedback from this informal user testing was, however, disappointing. The users experienced severe difficulties working with the new environment, but could not explain to us why. The users experienced two basic problems. First, the prototype version of ClockWorks v1.0 had bugs, leading to a sense of frustration. Most of these bugs arose when the tool was being used in ways not anticipated by the designers. Secondly, some aspects of the environment's design were confusing to the users. Since they could not understand how to use the tool to accomplish their goals, they found it difficult to articulate their problems.

From our experience, informal means of testing a user interface such as questionnaires and user surveys are useful in determining large questions about systems, such as gauging user satisfaction, but are not useful in finding detailed design problems. This is not surprising – visual programming environments are sophisticated tools supporting complex tasks. Often, the problems users have are not so simple that they can be described in a couple of sentences on a questionnaire, but instead are the result of failing to understand the conceptual model

of the tool, and of being unable to work out a strategy for how to perform their tasks. It is therefore difficult for users to express precise problems with a visual programming environment, other than that they found it difficult to use.

We have applied Gomoll's method for direct user observation [9] with much better results.

3.3.3 Gomoll's Ten Steps

The key idea behind Gomoll's ten steps is to have users test the system while being observed by testing experts. The users are given concrete, realistic tasks to carry out. Since the observers see the details of how the users attempt to carry out the tasks, they can understand the context of problems that arise. This form of testing can be done formally in a special purpose lab with video taping facilities to allow sessions to be analyzed later. As pointed out by Nielsen [5], however, useful results can also be obtained in informal circumstances. Gomoll's rules for user evaluation are sufficiently clear that even non-specialists can use them to obtain feedback on their designs.

To test ClockWorks v1.0, three Clock programmers were given a small program written in ClockWorks, and were asked to perform a series of modifications to the program. Each testing session lasted approximately one hour. The testing showed that the tasks identified in figure 5 were basically supported. The users were able to carry out the tasks they had been assigned. The testing showed some minor problems with the design of ClockWorks v1.0 which were relatively easy to fix. For example, the original design used menu commands for building architectures and for revealing and eliding information. These commands quickly became tedious for expert users, and were augmented by icons for fast browsing, and keyboard accelerators for building commands.

The testing also gave some hints about future problem areas. It was clear that the new environment provided far better support for incremental development, where designing and implementing an architecture could be interleaved. The user testing hinted that better support would be required for refining architectures and for reuse. These problems did not, however, become completely apparent until the environment was in production use, and real architecture refinement was taking place.

3.3.4 The Contribution of User Observation

User observation is an important part of evaluating user interfaces, particularly in identifying the many small issues that cause users to become confused. There are, however, two major limitations of applying user testing with a visual programming environment. First, users acting as subjects in the tests must already be fairly expert with the language; otherwise, it is hard to distinguish between errors in the design of the user interface and problems in learning what may be a complex programming language. This problem limited us to only three users who could act as test subjects. However, people who are experts with an earlier system may not use the new system in the way that was intended, instead carrying out the tasks as they would have with the earlier, less supportive system. For example, one of our users immediately exited the visual environment and started carrying out the required modifications using a text editor. (Interestingly, this user later became a complete convert to the visual environment.) Completely naive users have the advantage of having no preconceptions of the system. However, with a system as complex as a visual programming environment, naive users require so much time just to learn the system that it is unrealistic to expect to learn much about how the system will perform in production use.

The second challenge in applying user observation to visual programming languages is that for user observation to be successful, the tasks the users perform must be realistic. Serious problems, such as the weakness in support for refining architectures, only become apparent when complex tasks are being performed. However, if observation sessions are to be limited to a reasonable length of time, users don't have time to carry out complex programming tasks. Our approach of having the users modify an existing application seemed to be a reasonable compromise, since it combined tasks of understanding and refining applications.

From the testing performed according to Gomoll's ten steps and from subsequent use of the tool, we were able to develop the modified set of tasks of figure 6. These tasks more adequately represented the evolving incremental programming style of the Clock language. As it became apparent how the language was really going to be used, it became clear that ClockWorks v1.0 had serious usability problems in its support for understanding large

programs, and in refining programs. That these problems arose can be seen as a sign of success for ClockWorks v1.0, since it implied that users were now able to develop large programs. Ultimately, however, ClockWorks v1.0 was unsuccessful, since users preferred to return to textual notation for architecture refinement.

The following section shows how support was built into ClockWorks v2.0 to better support the tasks of figure 6. We believe that these ideas are of general interest to developers of visual programming environments: First, they illustrate that the success of a visual programming language depends on the success of its programming environment, not just on the expressiveness of the programming notation. The success of a visual programming environment in turn depends on its ability to support all facets of programming – building, understanding, and refinement of programs. Secondly, the features introduced into ClockWorks v2.0 demonstrate at least one successful approach to supporting these programming tasks.

4 Supporting the Identified Tasks

According to the task analysis of figure 6, the simplest activity that Clock programmers face is building a new architecture from scratch. To support this task, the programming environment must support the creation of architecture trees, and the definition of component interfaces.

The task analysis revealed that programmers tend not to have completely designed their architecture prior to beginning programming, but rather refine the architecture as they proceed. Programmers tend to keep their program working at all times, adding functionality step by step. To support this style of programming, special features are required in the programming environment to make it easy to introduce new architecture subtrees, to easily reuse different parts of architectures defined earlier, and to move around request handlers to reflect changes in required visibility. This incremental style of programming also requires support for easily restructuring programs; such support is required to easily split, combine and group components.

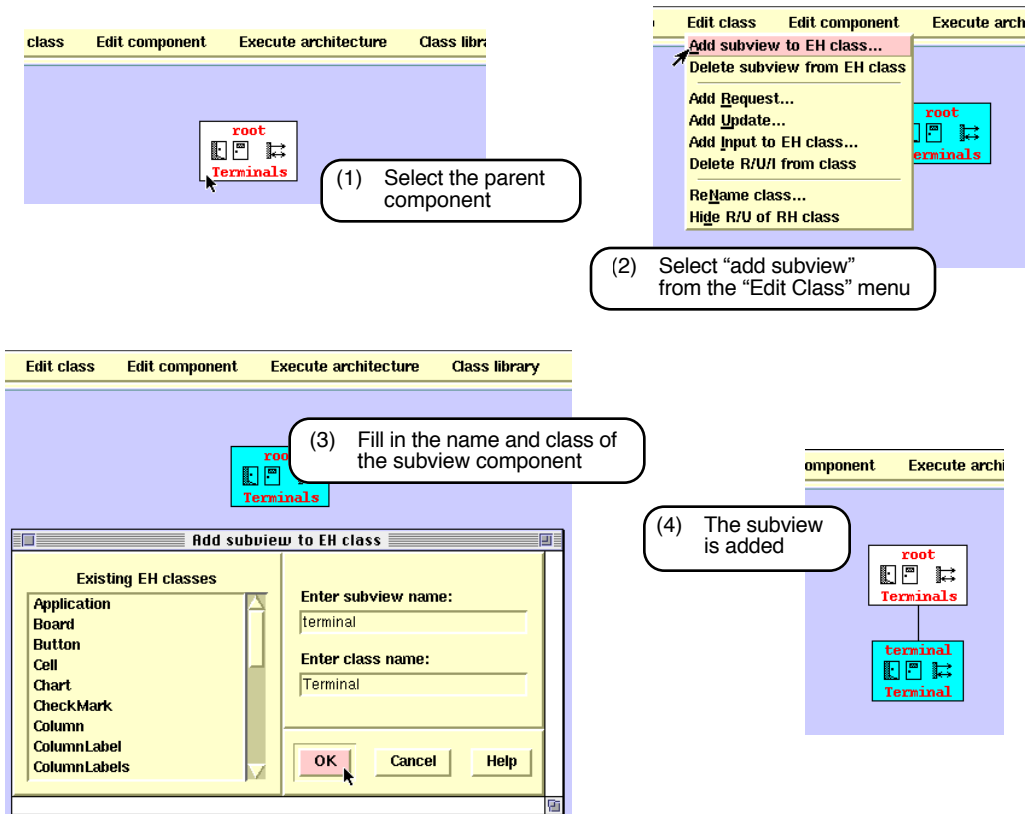


Figure 9: The steps involved in adding a new subview to a component.

Finally, programmers refer to the visual representation of programs in order to understand how the program works, and to verify that the program has the expected properties. To support these tasks, facilities are required to easily elide information, allowing programmers to concentrate on the parts of the program of most immediate interest.

We now show how the ClockWorks environment provides support for each of these three tasks of *building*, *refining* and *understanding* visual programs.

4.1 Building Architectures

Actually building new architectures is the area where Clock programmers spend the least part of their time. Large parts of programs can be built from existing components, found in libraries or in other programs. Most of the programming time is spent in refining and

modifying existing architectures.

Developing new programs must be supported in as straightforward a way as possible. In ClockWorks, architectures are built using simple, direct-manipulation commands. All commands are available in menus, and are also available via keyboard shortcuts.

To give the flavour of developing architectures in Clock, figure 9 shows how a subview can be added to an existing component. First, the programmer selects the component to receive a new subview (step 1 of figure 9). Next (step 2), the programmer selects “*Add Subview*” from the “*Edit Class*” menu. (Recall that adding a subview to any component of the *Terminals* class is considered to be a modification to the class itself.) A dialogue box (step 3) prompts the programmer to enter the name and class of the new subview. The programmer may choose an existing class, or define a new class simply by entering the new name. Finally, in step 4, the new subview is added to the selected component. All editing operations, including adding request handlers and defining the interface of a component, are performed in this way.

The definition of the “*Add Subview*” command exhibits one interesting feature that was added as a result of user observation. When the new subview is added, it becomes selected, and the parent ceases to be selected. The reason for this is that experienced programmers typically move straight from adding a subview to defining its interface, using keyboard shortcuts. Moving the selection to the new component allows the programmer to perform this efficient operation without an annoying intervening stage of moving from the keyboard to the mouse and back to the keyboard again. Design details as fine as this add a great deal to the usability of a system, but practically are only discovered if testing is performed with real users.

4.2 Refining Architectures

As pointed out by Burnett et al. [11], there are many problems hindering the scalability of visual programs beyond toy examples. Our experience indicates that one of the crucial issues in scalability is being able to easily modify programs. The task analysis of Clock programmers revealed that programmers do not come to the computer with a completely

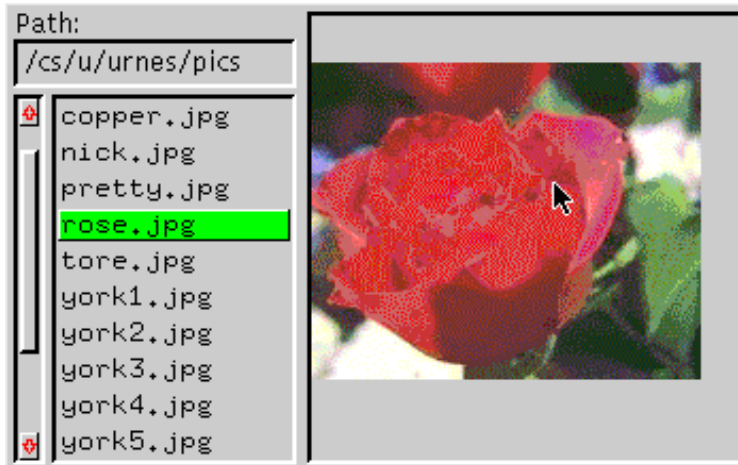


Figure 10: A simple JPEG image viewer written in Clock.

defined architecture, but rather develop the architecture interactively while working with ClockWorks. Programmers work incrementally, first developing a simple working skeleton of their program, then repeatedly modifying the program by fleshing out its details. At each stage of this incremental development, programmers start from a working version of the program, make changes, and then return to a working state before attempting the next change. Since Clock is intended for prototyping user interfaces, we were happy to see this form of incremental development, and wished to support it as well as possible.

To support incremental development of architectures, we needed to provide support for easily adding features to architectures, or *composing* new architectures with the existing one. Since architectures built incrementally can easily become unstructured, we also provide support for easily restructuring the architecture, with the emphasis on being able to easily modify the structure of architectures which have already been defined.

4.2.1 Composing Architectures

Figure 10 shows a simple JPEG image viewer prototyped in Clock. Users can select a directory, and then select an image from a list of file names. The image itself can be panned within the window by clicking on it and dragging.

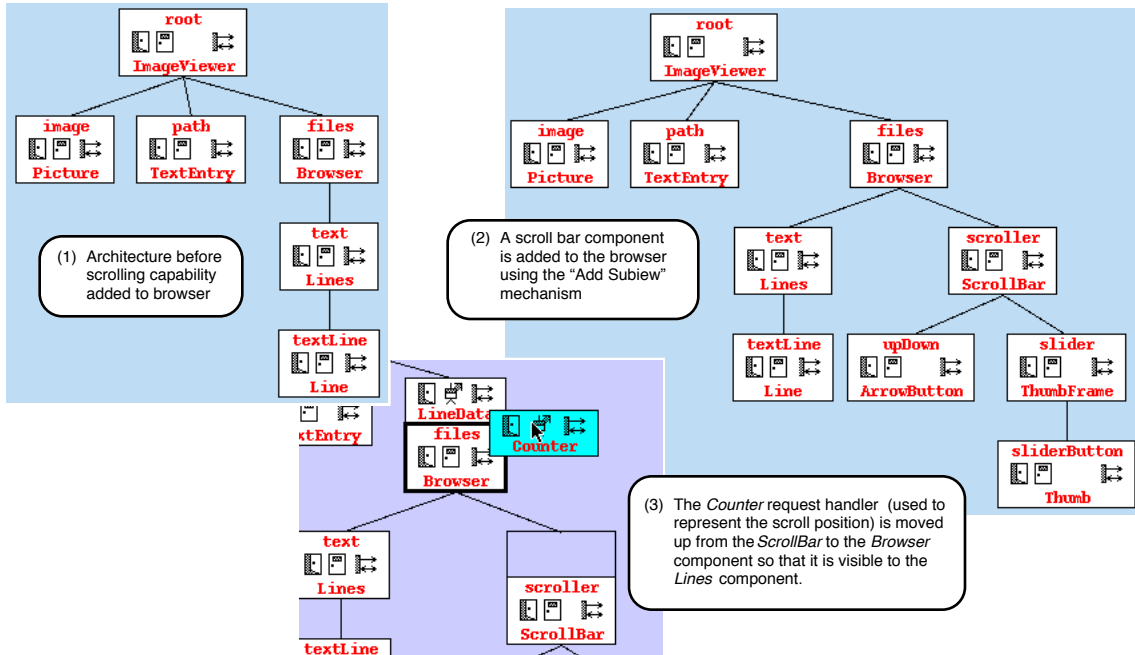


Figure 11: Incremental development of the JPEG image viewer from figure 10.

Figure 11 shows how this program can be developed incrementally. Step one shows a version of the image viewer architecture. Here, the image viewer consists of a *Picture* class implementing the view of the image itself, a *TextEntry* class implementing the entry field for the path name, and a *Browser* class implementing the list of files containing images. In this version, the programmer has not yet implemented the scroll bar adjacent to the list of names.

Step two of figure 11 shows that to add a scroll bar, the programmer need only add a scroll bar subview to the *Browser* class. The components making up this subview are added by repeatedly using the “*Add Subview*” mechanism that was seen in figure 9. (The code implementing the *Browser* component must of course also be modified to correctly make use of the scroll bar.)

Step three shows the final stage required to integrate the scroll bar into the architecture. The scroll bar uses a *Counter* request handler to represent the current position of the scroll bar. The *Lines* component must be able to refer to this value in order to determine which

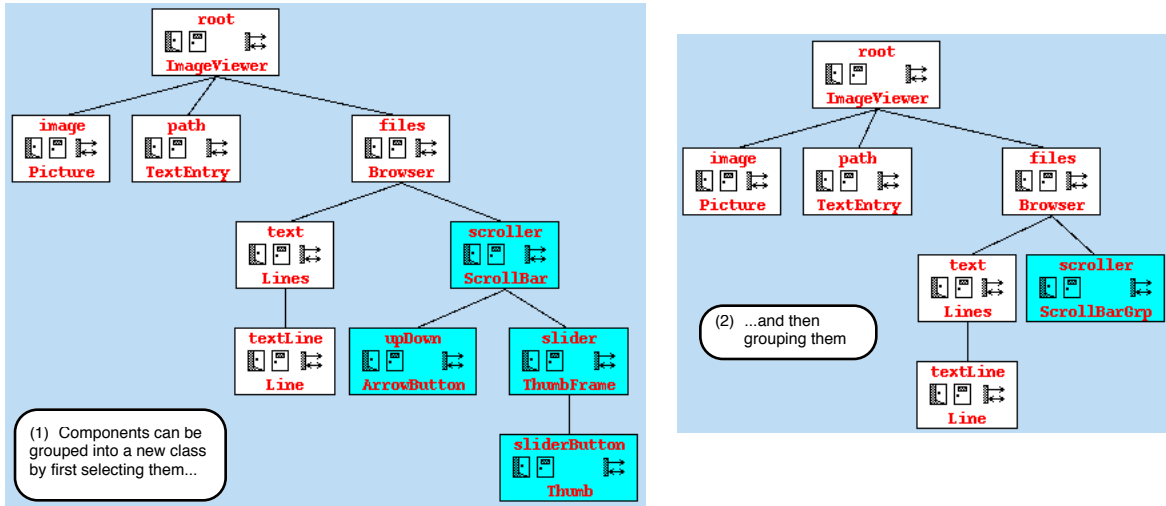



Figure 12: Architectures can be refined by introducing groups. This is usually done incrementally, after the architecture is defined.

lines currently appear on the display. Therefore, the *Counter* is moved up one level in the architecture. As was specified by the UAN of figure 8, this is accomplished simply by clicking on the “” icon on the counter, and moving it over its new location in the architecture tree.

This example shows how by providing easy support for adding new subviews in an architecture and for moving request handlers in the tree, we are able to provide good support for incremental development of architectures.

4.2.2 Restructuring Architectures

As architectures are developed incrementally, programmers will often wish to restructure the architecture to make it easier to understand and easier to modify. Programmers also wish to structure groups of components to make them easier to reuse in other contexts.

The major restructuring operations that Clock programmers perform are:

- *Splitting and combining* components. Components are meant to handle a specific task. Sometimes it becomes clear that a group of components is performing one logical task

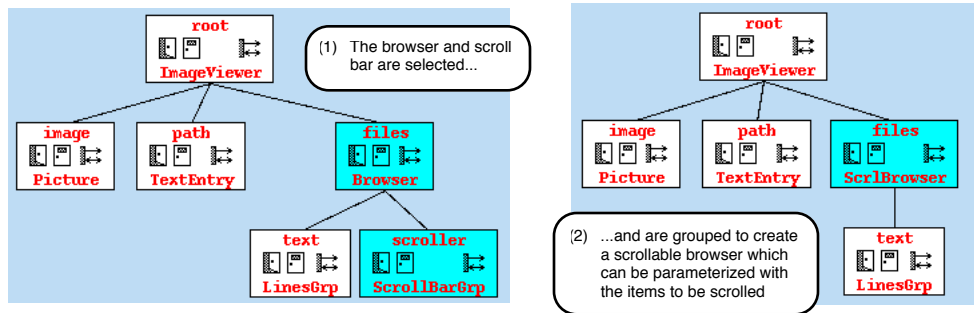


Figure 13: Group classes may themselves take subviews. Here the scroll bar and the browser are combined to create a general scrollable browser whose subview is the items to be scrolled.

and should be combined. Sometimes it is clear that a single component is performing more than one task and should be split into two.

- *Introducing new levels* in the architecture. Sometimes a single component combines many subviews, some of which may be logically related. In this case, new levels can be introduced in the architecture tree to combine the related subviews.
- *Moving request handlers*. The position of request handlers in the tree represents the visibility of the request handlers. Sometimes request handlers may be moved up or down the tree to more logically reflect their scope.
- *Structuring for reuse*. Component reuse is a very important part of programming in Clock. Sometimes a programmer will note that a group of components fulfill a logical function that occurs in many architectures. Programmers will restructure the component group to remove details specific to the current architecture, and allow easy reuse in the future.

In fact, not all of these tasks are supported well in the current version of ClockWorks. ClockWorks does provide good support for splitting and combining components, moving request handlers, and structuring for reuse. Currently, moving entire component subtrees is possible, but cumbersome. As the system continues to evolve, better support for these tasks will be introduced.

Figure 12 shows how a group of components can be combined to create a new class. In

the architecture for the image viewer, we note that the scroll bar consists of a group of components. Scroll bars are a useful interaction technique that can be reused in many contexts. Instantiating each of the many classes used in building a scroll bar is tedious, however, so we wish to define a single *ScrollBarGrp* class which stands for the scroll bar architecture. To group components, we simply select them (step 1 of figure 12), and then use the *Group* menu command to group them (step 2 of figure 12). The *ScrollBarGrp* class is now defined, and can be placed in any architecture just like any other class.

Figure 13 shows that group classes can take subview parameters. Here, we decide to create a class implementing a scrollable list of arbitrary items. The scroll bar is combined with the *Browser* component to create a new *ScrlBrowser* class. This class takes a subview which implements the items to be scrolled (here a list of files containing images.) In other architectures, other components can be used for this subview.

In practical programs, this grouping mechanism is used extensively to simplify the structure of architectures. Since groups can be arbitrarily nested, programs consisting of hundreds of components can appear to have quite straightforward structure. Since groups can be easily defined after the architecture has been constructed, programmers are encouraged to maintain reasonable structure in their systems. Since ungrouping and regrouping is easy, programmers are encouraged to create elegant group definitions, aiding the development of reusable component libraries.

In summary, Clock programmers develop programs in an incremental fashion. To support this incremental development, the programming environment must make it easy to add to programs, and to restructure programs as their design evolves.

4.3 Understanding Architectures

One of the early problems in Clock was information overload. As programs became larger, programmers became less able to keep track of which components issued or implemented what updates and requests. As architectures evolved, keeping track of this information soon became unmanageable. Architectures needed only to reach sizes of a few dozen components before becoming completely unmaintainable.

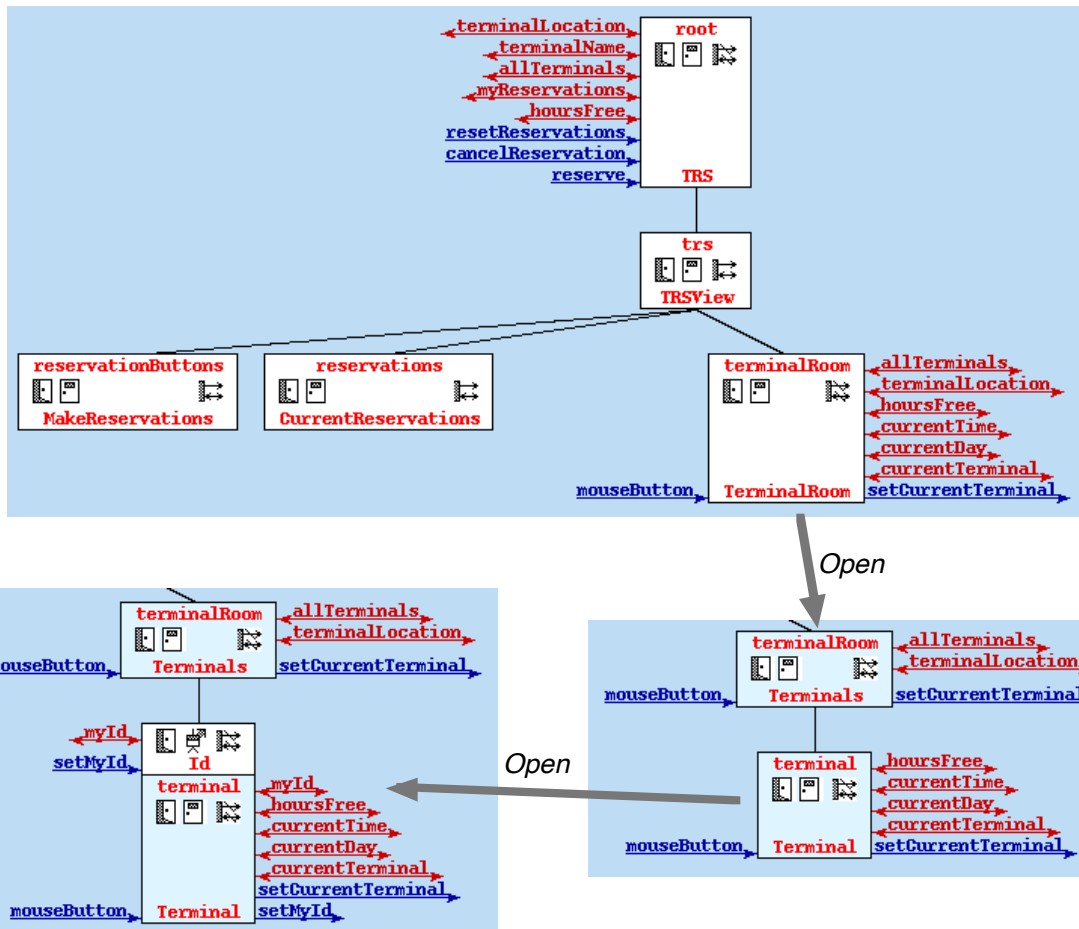


Figure 14: The structure of architectures can be opened up by clicking the “open door” icon on components. Interfaces can be toggled by clicking the “arrows” icon.

The first version of ClockWorks provided visual support for viewing the structure and interfaces of components. This support increased the effective size of manageable programs. However, programs still could not be very large before becoming difficult to understand and modify.

The current version of ClockWorks demonstrates that the key to scaling visual languages is in providing good support for *hiding* information. Programmers need to be able to easily obtain the information they need to understand some aspect of their program, but should not be forced to view any more information than is necessary. This philosophy simultaneously addresses the problems of information overload and of limited display real-estate.

Our view in ClockWorks is that programs are not a single static notation as in textual programming, but are rather a model. Adding to programs is entering more information into the model. Viewing a program is projecting some aspect of the model onto the display. Commands within a visual programming environment should therefore make it easy to view whatever aspects of the model that are currently of interest to the programmer.

In ClockWorks, facilities are provided for selecting the detail at which the *structure* of a program and the *communication* between program components is to be viewed. As seen in figure 14, each component has an “open” and “close” door icon. Clicking on the open door always means “show more detail”; similarly, clicking on the close door means “show less detail”. The architecture of figure 14 implements the terminal reservation system of figure 1. When the programmer opens the *TerminalRoom* component, it is revealed to be a group consisting of the components *Terminals* and *Terminal*. When *Terminal* is opened, it is shown to contain the single request handler *Id*. If *Terminal* were opened once again, the code implementing the component would be shown. If the programmer were to close these views, the path would be retraced, showing successively less detail. This simple opening and closing mechanism allows the programmer to quickly obtain detail in some parts of the system, while maintaining only a high-level overview in other parts.

The third icon on each component allows the programmer to toggle the display of the component’s interface. Interfaces show what updates and requests a component is capable

of handling, or may make in performing its own task. Programmers can quickly understand how different parts of the program communicate by examining their interfaces.

Elision of information in interfaces is crucial to keeping displays small enough to fit on a screen, and for reducing the amount of information a programmer must consider at any one time. ClockWorks provides both automatic and manual elision facilities. For example, in figure 14 the interface of the *Terminal* component shows only those updates and requests that are not handled internally by the group. Internal requests (such as *myId*) are not shown in the group interface. Programmers may also choose to further restrict the information shown in the interface, manually overriding the automatically calculated interface.

4.4 Treating Visual Programming Languages as User Interfaces

This section has demonstrated that once the task analysis of figure 6 was available, it was clear what facilities were required in the visual programming environment. The resulting ClockWorks v2.0 environment is sufficiently successful that the textual representation of Clock architectures is no longer used. In fact, new users of Clock do not need to be aware that the textual form even exists. Our experiences show that performing a task analysis of programmers using the language helps reveal where programmers are really spending their time, and therefore helps to show what support is required in the programming environment. Iterative refinement of visual programming environments is also crucial, since as the environment improves, programmers will begin to use the language in different, more effective ways. Controlled testing with users is crucial to discovering what unforeseen problems may result in the programming environment.

5 Conclusions

This paper has described the *ClockWorks* environment, an environment for the visual programming of software architectures. We developed ClockWorks using iterative refinement, and applied user interface evaluation techniques to determine how successfully ClockWorks supported programmers' tasks. It is crucial to our approach that we began by performing a task analysis of Clock programmers. This task analysis revealed to us how programmers

actually use our language, and therefore what programming operations are most important to support. Our conclusion was that to be successful, a visual programming environment must not only provide good support for building and viewing programs, but also for refining programs as they grow in size. We showed how facilities in ClockWorks support the modification and restructuring of visual programs as they are developed incrementally.

Acknowledgements

Clock and *ClockWorks* were developed by the authors, Roy Nejabi and Gekun Song. This work was partially supported by the Natural Sciences and Engineering Research Council, the Information Technology Research Centre, and the Royal Norwegian Research Council.

References

1. T. C. N. Graham (1995) *Declarative Development of Interactive Systems*, volume 243 of *Berichte der GMD*. R. Oldenbourg Verlag.
2. T. C. N. Graham & T. Urnes (1996) Linguistic support for the evolutionary design of software architectures. In *Proceedings of the Eighteenth International Conference on Software Engineering (ICSE'18)*. ACM Press.
3. P. Hudak & P. Wadler (1991) Report on the functional programming language Haskell (v1.1). Technical Report YALEU/DCS/RR777, Yale University.
4. P. Booth (1989) *An Introduction to Human-Computer Interaction*. Lawrence Erlbaum Associates.
5. J. Nielsen (1993) *Usability Engineering*. AP Professional, Cambridge, MA.
6. R. Jeffries, J. R. Miller, C. Wharton & K. Uyeda (1991) User interface evaluation in the real world: A comparison of four techniques. In *ACM SIGCHI 1991*, pp 119–124.
7. H. R. Hartson, A. C. Siochi & D. Hix (1990) The UAN: A user-oriented representation for direct manipulation interface designs. *ACM Transactions on Information Systems*, **8**(3):181–203.
8. C. Morton (1994) Tool support for component-based programming. Technical Report CS-94-02, Department of Computer Science, York University.
9. K. Gomoll (1990) Some techniques for observing users. In B. Laurel (ed.), *The Art of Human-Computer Interface Design*, pp 85–90. Addison Wesley.
10. J. R. Cordy (1992) Hints on the design of user interface language features – lessons from the design of turing. In B. A. Myers (ed.), *Languages for Developing User Interfaces*, chapter 18, pp 329–340. Jones and Bartlett.
11. M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang & P. van Zee (1995) Scaling up visual languages. *IEEE Computer*, **28**(3):45–54.