Viewpoints Supporting the Development of Interactive Software

T.C. Nicholas Graham

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

Abstract

The use of a variety of software viewpoints is helpful in the user-centered development of interactive software. Viewpoints of interactive software include paper and pencil mockups, task-oriented specifications, architecture views, and code views. In our experience, however, programmers prefer to develop user interfaces using prototyping tools that emphasize the code view only, potentially resulting in both a lower quality of user interface and a poor quality of delivered code. This paper surveys the role of different software viewpoints in a user-centered development process. The paper argues that to be successful, such a process must support incremental development, easy movement between viewpoints, and good tool support for manipulating different viewpoints.

Keywords: user-centered design, software viewpoints, user interface toolkits, groupware development

1 INTRODUCTION

Interactive software permits people to interact with a computer to perform some task. To make computer use more natural and available to a broader population, much research effort has been expended into how to create interactive software that is easier to use, for example through the use of graphical user interfaces. Modern interactive applications may involve multiple media such as text, graphics, animation, sound and video, and may support the collaborative work of multiple users.

The development of interactive applications differs from that of traditional software in that it is highly experimental – it is hard to determine the usability of a system before it is implemented and tested with real users. Consequently, the design of an interactive application evolves considerably throughout its development as parts of the design are implemented, tested and redesigned. The traditional waterfall model of software development therefore does not work well with interactive software, since this model assumes we know what we are going to implement before implementation starts.

In response to the experimental nature of user interface development, a variety of *user centered* development methods have been proposed [2]. The key recommendations of these methods are: make the design process *participatory*, where end users take place in the design team along with graphic designers, domain experts and programmers; perform detailed user needs analysis prior to design; evaluate the user interface as best as possible before implementation, and plan to iteratively refine the design after testing with users. These methods can also be seen as development through the use of viewpoints – for example, graphic designers use paper and pencil mockups; domain experts use task-oriented specifications, and programmers use code.

User centered development methods are predicated on the assumption that implementation is expensive, and that the overhead of design and evaluation will therefore be amortized by reduced implementation costs. However, new tools for user interface development such as Tcl/Tk [9], Visual Basic [7] and Macromedia Director have greatly reduced the cost of developing user interface prototypes. These tools are typically visually oriented and interpretive, allowing quick experimentation with new ideas. The speed of modern computers has meant that such tools usually give adequate performance for production purposes.

The result of such tools is that programmers have become less willing to consider viewpoints other than code. In a tool like Visual Basic, for example, it is almost as easy to implement a screen layout as to draw it on paper. Programmers use tools to help flesh out the design of a user interface, rapidly iterating between testing and refining ideas. Programmers therefore see structured methodologies for user interface development as cumbersome and slow to produce results. The resulting tool-centered approach to user interface design has, however, some serious problems:

Programmer in control: When the development process is centered around code development, the programmer becomes the owner of the process. This discourages participatory design, eventually leading to poorer quality user interfaces.

Unmaintainable code: Rapid development tools tend to sacrifice structure for flexibility. Programs developed in an experimental, evolutionary style tend to become unmanageable. Products may execute fast enough for production purposes, but may be unmaintainable.

Poor linkage to task analysis: Tool-based processes do not build in appropriate analysis of the user interface design with respect to the tasks being supported. The risk is high of developing a product that does not actually perform what the user group needs.

Poor support for modern user interfaces: Modern user interfaces include such features as multiple users and multiple media. To support these features requires programming con-

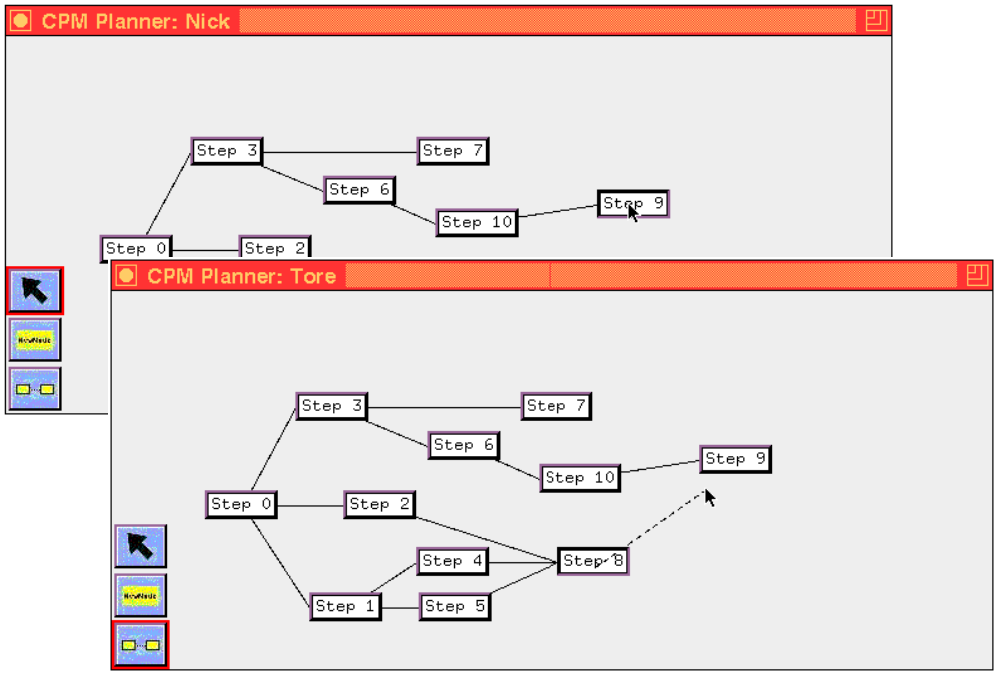


Figure 1: A critical path planner application implemented in Clock. Nick is moving a node while Tore is connecting two nodes.

currency, concurrency control, distribution and networking. These features are not handled well by current rapid development tools.

We have been researching a development process based on combining the early evaluation and participatory nature of the user-centered approaches with the flexibility of the tool-based approach. The key principles behind this process are:

- The various participants in the process should be able to work in the representation they find most natural, *at any time*;
- It must be easy to move back and forth between viewpoints.

In order to achieve this, we have found it necessary that:

- All notations must support an incremental development style, where information can be added easily, and where sub-descriptions can be easily composed;
- Related notations may differ in their points of view *or* in their level of detail, but not both;
- Excellent tool support must be provided to encourage the development team to keep views synchronized.

The next section briefly surveys the viewpoints we have found to be useful in user-centered design. The following section then argues why we believe an incremental approach to be necessary in using these different viewpoints, and discusses the importance of excellent tool support for the different viewpoints.

2 VIEWPOINTS AND USER-CENTERED DESIGN

Figure 2 shows the outline of our user-centered design process. Each phase in the process works with a different representation

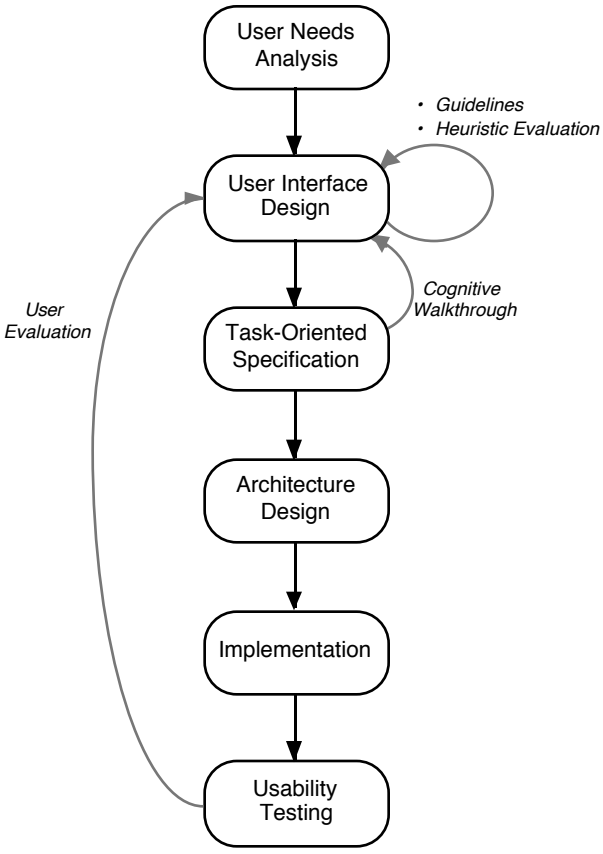
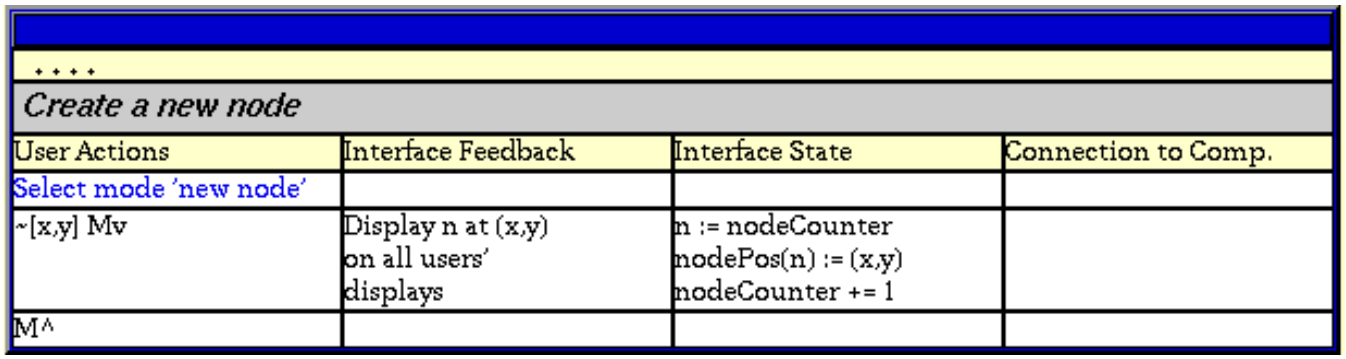


Figure 2: A User-Centered Design Process.



Create a new node

User Actions	Interface Feedback	Interface State	Connection to Comp.
Select mode 'new node'			
~[x,y] Mv	Display n at (x,y) on all users' displays	n := nodeCounter nodePos(n) := (x,y) nodeCounter += 1	
M^			

Figure 3: Task-Oriented Viewpoint using the User Action Notation [6].

(or viewpoint) of the eventual system. These viewpoints carry different information, and are appropriate for use by different participants in the design process. As presented in this diagram, this process is similar to other user-centered design approaches [2], and in particular Boehm's spiral model [1]. This process is called the *Clock methodology*, in recognition of its basis in the Clock programming language for interactive systems.

The following sections briefly describe the stages in this process and the viewpoints used by the developers at each stage. To help illustrate the process, we use a simplified multiuser program for collaborative project planning. As shown in figure 1, multiple users can add nodes to a critical path network, connect them, and move them. In a more realistic version of the program, users can also attribute times and costs to the nodes, allowing them to experiment with trade-offs in resource allocation.

2.1 User Needs Analysis

This phase is concerned with characterizing the intended user group of the system, and detailing the tasks that the system is intended to support. Deliverables from this phase include a user characterization and a hierarchical task analysis. Contributors to this phase should include domain specialists, including members of the intended user group.

2.2 User Interface Design

Here the design of the user interface is specified, typically based on paper and pencil mockups and prose description. Contributors to this phase should include user interface specialists and graphic designers.

2.3 Task-Oriented Specification

In this phase, the user interface is evaluated with respect to the tasks that are to be supported. Analysts demonstrate how each of the users' tasks can be accomplished using the proposed system, evaluating the completeness, the ease of use and the consistency of the user interface. Contributors to this phase include user interface specialists and domain specialists. In this phase, we use Hartson, Siochi and Hix's *User Action Notation* [6].

Figure 3 shows an example of a UAN specification of how a user adds a node to the network of figure 1. The *user actions* column of the chart shows that to create a new node, a user must first select the "new node" mode (by clicking on the new node button), then move to some position on the screen (`~[x,y]`), depress the mouse button (`Mv`), and then release the button (`M^`). The interface feedback and interface state columns are used to record the effects of each action on the display and on the internal state of the user interface. This form of specification is called *task-oriented*, since it describes how each of the user's tasks is carried out in the system.

Figure 3 is a screen snapshot taken from the Clock UAN browser [11]. This browser allows UAN specifications to be viewed in a hypertext form; for example, clicking on the "select mode" subtask would bring up the UAN chart showing how a user changes modes.

2.4 Architecture Design

The architecture design of the system decomposes the system into components and specifies how the components communicate. Figure 4 shows one view of the architecture of the planning program of figure 1, encoded in the Clock architecture language [5]. The architecture is developed by a programmer, based on information in the user interface design and the task-oriented specification. In Clock, architecture designs follow the hierarchical structure of programs; for example, a network is composed of nodes, edges and (possibly) a rubber-band line.

Figure 4 is a screen snapshot of the *ClockWorks* [4] environment used to build and browse Clock architectures.

2.5 Implementation

The components of the user interface must themselves be implemented, providing a code view of the system. Figure 5 shows the code used to handle mouse input in the *CPMNetwork* component of figure 4. This code in fact implements the task of creating a node, as described in figure 3. The code states that if this component receives a "mouse button down" input event, then a new node is to be added to the network, positioned at the current mouse position, and the node id counter is to be incremented. This code is clearly similar to the UAN specification of figure 3, but is from the point of view of the system. The UAN specification describes how a user adds a new node to the network, while the code specifies how the system is to respond to mouse button input.

2.6 Usability Testing

The final phase of the process involves testing the system with real users, in order to identify user interface design errors. The results from this testing are used to revise the user interface design,

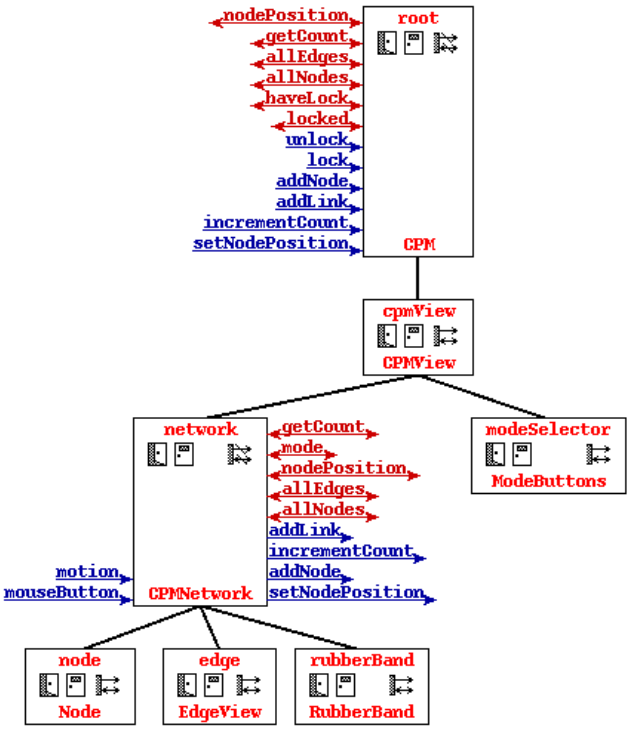


Figure 4: Architecture viewpoint showing application structure.

```
mouseButtonUpdt "Down" =
   if mode = AddingNode then
      % Add a new node at this position
      let id = numstr getCount in
         all [
            addNode id,
            setNodePosition id mousePosition,
            incrementCount
         ]
      end let
   else
      % In any other mode, has no effect
      noUpdate
   end if.

mouseButtonUpdt "Up" = noUpdate.
```

Figure 5: Code viewpoint showing implementation details.

propagating changes through the task-oriented specification and implementation.

3 AN INCREMENTAL APPROACH

The process described in the last section appears to engender an inflexible order of operations – first the user needs analysis is performed, then the design, then the task-oriented specification, followed by the implementation. Developers, however, are unwilling to follow such a structured process. For example, fine distinctions in a task analysis might be brought out in the presence of a quickly prototyped system. A proposed design might be easier to implement in prototype form than to describe using pictures and prose. Parts of a system may be tested with users prior to evaluation with a task-oriented specification.

As pointed out by Parnas and Clements [10], the quality of delivered software can be improved by giving the appearance of a rational software development process, even when it is impractical to actually follow one. That is, as long as a full user needs analysis is performed, and as long the the user interface design is fully evaluated through task-oriented specification and usability testing, it is unimportant in what order these steps were actually performed.

For the development of interactive systems, we advocate an incremental approach where different participants in the development operate in parallel, in the notation most appropriate to their role. For example, one of the strengths of the UAN notation is that it is often possible to specify each task in isolation, without having to worry about details of task interleaving or concurrency. It is therefore highly helpful to develop a UAN specification as a step towards implementation. Alternatively, when we wish to experiment with a variety of possible designs, it can be tedious to develop the UAN specification for each one prior to implementation. Instead, the designs can all be coded directly, and the UAN filled in afterwards for the chosen design.

One crucial benefit of the incremental approach is that ownership of the development process is distributed. Since developers can work in parallel, each using their own representation of the system, information can flow in any direction between implementation, testing and design.

4 EXPERIENCE

We have experimented with variants of this methodology for four years in a senior undergraduate course at York University, allowing over 100 students to build over 30 interactive systems. Additionally, we have performed three substantial case studies within our research group [3, 8, 11].

Our major observations from these experiences are:

Tool support is critical: Programmers can be convinced to perform one iteration of developing a viewpoint if they believe it will help them towards implementation. However, once implementation has commenced, it is very difficult to convince them to *maintain* alternative viewpoints. Excellent tool support for quickly updating and modifying viewpoints is essential. In our experience, it is very difficult to convince programmers to maintain their UAN specifications throughout the revisions their programs undergo. To help alleviate this problem, Eric Telford has developed a UAN browser [11], dramatically easing the task of working with UAN specifications.

Viewpoints cannot differ too greatly: Representations of software may differ in their point of view, or in their level of de-

tail, but not both. For example, our early experiments with moving from UAN specifications to X/Motif code proved unsuccessful, since the gap between the two viewpoints was too large. Inserting the stage of developing a Clock architecture for the system led to far more successful results. As can be seen by comparing figures 3 and 5, the Clock code carries a similar level of detail to the UAN code, but from a different point of view.

5 CONCLUSIONS

This paper has discussed the role of software viewpoints in a user-centered process for user interface development. The paper has argued that different viewpoints can aid in participatory design of user interfaces by supporting developers with different roles, and distributing ownership of the process. We have also argued that multiple viewpoints are best used in an incremental fashion, where information flows between the different representations. In order to support this incremental approach, excellent tool support is required.

6 ACKNOWLEDGEMENTS

Clock and *ClockWorks* were developed by the author, Tore Urnes, Catherine Morton, Roy Nejabi and Gekun Song. The Clock UAN browser was developed by Eric Telford. This work was carried out at the York Laboratory for Computer Systems Research, and was partially supported by the Natural Sciences and Engineering Research Council and the Information Technology Research Centre.

References

[1] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(2):61–72, May 1988.

[2] Judy Brown. Methodologies for the creation of user interfaces. Technical Report CS-TR-96-1, Department of Computer Science, University of Victoria at Wellington, 1996.

[3] Herbert Damker. Spezifizierung und Architekturentwurf von Benutzungsschnittstellen: eine Fallstudie in der Clock-Methodologie. Studienarbeit, Universität Karlsruhe, September 1992.

[4] T.C. Nicholas Graham, Catherine A. Morton, and Tore Urnes. Clockworks: Visual programming of component-based software architectures. *Journal of Visual Languages and Computing*, July 1996. (To appear).

[5] T.C. Nicholas Graham and Tore Urnes. Linguistic support for the evolutionary design of software architectures. In *Proceedings of the Eighteenth International Conference on Software Engineering (ICSE'18)*, pages 418–427. IEEE Press, March 1996.

[6] 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*, 8(3):181–203, July 1990.

[7] Microsoft Corp. *Visual Basic User Manual*, 1994.

[8] Catherine Morton. Tool support for component-based programming. Technical Report CS-94-02, Department of Computer Science, York University, May 1994.

[9] John K. Ousterhout. An X11 toolkit based on the Tcl language. In *Proceedings of the 1991 USENIX Winter Conference*, pages 105–115, January 1991.

[10] David L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, February 1986.

[11] Eric Telford. Developing a UAN browser in clockworks: a case study of incremental development using the clock methodology. Technical Report CS-96-03, Department of Computer Science, York University, June 1996.