# Linguistic Support for Reuse in the Development of User Interfaces

Khuzaima S. Daudjee
Email: daudjee@cs.yorku.ca

T.C. Nicholas Graham
Email: graham@cs.yorku.ca

Dept. of Computer Science
York University
4700 Keele St.
North York, Ontario
CANADA M3J 1P3

**Abstract**

User interface development differs from that of traditional software. This paper discusses the special linguistic features necessary for supporting reuse in the development of graphical user interfaces. It is shown how limiting the domain of reuse to the user interface leads to stronger results than can be achieved for the general reuse case. These ideas have been demonstrated in the *Clock* system [1], a language for the rapid prototyping of graphical user interfaces.

**Keywords:** Rapid prototyping, user interface, object-oriented framework, Clock methodology, interactive system.

**Workshop Goals:** Learning; establish better communication with other researchers; exposure to other methods of reuse.

**Working Groups:** Reuse and OO methods, Tools and environments, Reusable component certification, Reuse and formal methods.

# 1 Background

**Khuzaima S. Daudjee** is interested in Software Reuse from various perspectives. He investigated reuse in software systems from a heuristic search standpoint as a graduate course project. His current work is aimed toward reuse based on an object-oriented framework in the Clock language.

**Nicholas Graham** has an extensive background in programming language design and implementation. His most recent work has been the development of *Clock*, a language for the rapid prototyping of interactive systems. Clock is intended to support the high-level programming of user interfaces through reuse of a palette of predefined components. The language has been designed to best support the creation of flexible, reusable components.

# 2 Position

Developing modern interactive applications requires a different approach from that of traditional software development. Good user interfaces cannot be designed *a priori*, but must be refined through iterations of rapid prototyping and user testing. The high cost of such iterative design of user interfaces has led to a need for high-level tools that support the rapid prototyping of interactive software.

Reuse aids the rapid prototyping of interactive software by raising the level of the basic building blocks used to construct a system. Programmers work with primitives such as menus, radio buttons and dialogue boxes, rather than variables and pointers. This high-level support not only saves the time required to program the individual components, but also helps the programmer in determining a high-level structure (or *architecture*) for the program. Reuse of existing user interface components additionally helps in enforcing user interface consistency, and adherance to standards.

Our approach to reuse has been to provide linguistic and architectural support for creating reusable components. This paper briefly discusses the major ideas behind this approach. Our current research focuses on building libraries of reusable components, and investigating a development methodology based on reuse.

## 2.1 Linguistic Support for Reuse in User Interfaces

The process of user interface development can differ from the development of traditional software. These differences lead to special requirements for how support for reuse should be built into user interface development languages. Our research has identified three key requirements:

- Reuse should be based on a black box model;
- Reasonable defaults should be provided for component parameters;
- Support should exist for combining existing components into a user interface architecture.

### 2.1.1 Black Box Model

In traditional software development, the design of reusable components is highly challenging. For example, a symbol table designed for one compiler would almost certainly not be directly usable in another. Even when a programmer attempts to design a component for reuse, it is very difficult to anticipate every use to which the component will be put, and to provide all the necessary degrees of freedom in instantiating the component [2]. This has led to an approach called *white box* reusability [3], where components are designed for reuse as best as possible, but where an adaptation process still requires code-level customization of the component.

In contrast, graphical user interfaces typically consist of a relatively small number of comparitively well-understood components. It is reasonable to expect to define the basic components of user interface development, such as menus, buttons, dialogue boxes and scroll bars, in a *black box* fashion where the components do not have to be modified to be reused.

### 2.1.2 Defaults

One of the problems with black box reuse is that components can have an overwhelming number of parameters which must be instantiated before the component can be used. Understanding all the parameters and instantiating them can place an unreasonable burden on the component user. Consider, for example, a component implementing a *radio button* behaviour (radio buttons have the property that exactly one of a set of buttons can be selected at a given time.) Potential parameters include: the buttons themselves, which button is initially selected, how the buttons are to be positioned, and how the component is to report the selection of a new button.

In a rapid prototyping context, the programmer may not be initially interested in the details of all of these parameters, but may be willing to put up with some reasonable, but possibly incorrect, default value. For example, a programmer might wish to lay out a list of radio buttons in a circular clock face, but be willing to accept a horizontal layout as a first approximation. It is therefore important for a language to permit the attribution of default values to components, and to provide an easy mechanism for overriding the defaults.

In traditional software, the automatic use of defaults may lead to incorrect behaviour that is hard to track down. In user interfaces, however, the incorrect nature of a default will usually be immediately visible on the display: a button may be square instead of rounded, a menu might be in the wrong font, an alert box may not display the correct message. Such details are often unimportant in the early versions of a user interface, and can be specialized at any time.

### 2.1.3 Support for Architecture Design

The use of predefined components raises the level of user interface construction. Ideally, user interfaces could be constructed completely by connecting together predefined components. This approach not only saves the time of recoding the reused components, but also aids in the design of user interface architectures. In general, user interface architectures must support ease of modification and possess clear separation of concerns and communication structures. Such design is difficult to achieve, especially in a rapid-prototyping context where the user interface itself is not fully specified before implementation begins. The availability of well-designed, high-level components aids in creating a well-structured architecture.

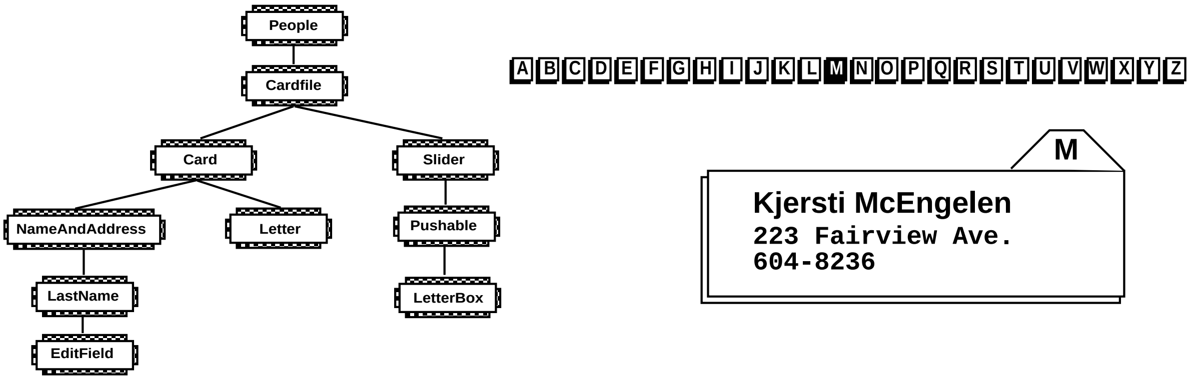


Figure 1: A Clock architecture for a card file name and address program, and the resulting user interface

## 2.2 Our Approach

The previous section identified three key requirements for reuse mechanisms in the development of user interfaces. Within the *Clock* [1] language, we attempt to meet each of these three requirements. Clock programs consist of a tree of components, structured in an object-oriented style. Components themselves are coded in a high-level, purely functional language. Figure 2.2 shows an example of a name and address card file program, and the Clock architecture that implements it.

Communication between components is based on a mechanism of updates and requests. These may only travel up the tree, meaning that components can know what data exists above them, but not below. This leads to a general property where component subtrees can be replaced by any other subtree matching the same request/update interface, while being guaranteed not to require changes to any other part of the architecture.

Predefined components are designed to be used in a black-box fashion. For example, the *Pushable* component adds a push-button behaviour to whatever object appears below it in the tree. Components may be parameterized. The language provides a natural mechanism for setting defaults for these parameters. The defaults may then be overridden at either the architecture or the program level.

Clock provides a mechanism of *invariant functions* that allows components to specify consistency conditions that are automatically maintained. These invariant functions simplify the communication structures between components, making it easier to combine and replace existing components in an architecture.

## 2.3 Comparison

A number of user interface toolkits support some form of reuse. InterViews [4] is a toolkit based on an object-oriented framework [5]. InterViews supports object categories where each category is a hierarchical structure of object classes within a common class. Reuse is facilitated by allowing objects to be composed from various subclasses. InterViews is intended for producing production quality user interfaces, and therefore does not include higher-level features used in Clock to ease reuse. In particular, InterViews has no equivalent to Clock's invariant functions.

Garnet [6] is a toolkit intended for rapid prototyping of user interfaces. Garnet provides *interactors* based on the MVC [7] model, where the interactors correspond to the controller, the object-oriented graphics correspond to the view, and the code is the model. Constraints in Garnet are used to link the model, view and controller parts, giving a functionality similar to Clock's invariants. The inheritance mechanism can be used to set and override default values on components. Garnet differs from Clock in that Garnet is a more general system, where structuring programs for reuse is a matter of discipline; in Clock, the means of structuring architectures is built into the language.

## References


[1] T. N. Graham, "Constructing user interfaces with functions and temporal constraints," in *Languages for Developing User Interfaces* (B. A. Myers, ed.), Jones and Bartlett, 1992.

[2] R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 2, pp. 22–35, June/July 1988.

[3] C. W. Krueger, "Software Reuse," *ACM Computing Surveys*, vol. 24, June 1992.

[4] M. Linton, J. Vlissides, and P. Calder, "Composing User Interfaces with InterViews," *IEEE Computer*, pp. 8–22, February 1989.

[5] R. E. Johnson and V. F. Russo, "Reusing Object-Oriented Designs," Tech. Rep. UIUCDCS 91-1696, University of Illinois, May 1991.

[6] B. Myers, D. Guise, R. Dannenberg, B. V. Zanden, D. Kosbie, P. Marchal, and E. Pervin, "Comprehensive Support for Graphical, Highly Interactive User Interfaces: the Garnet user interface development environment," *IEEE Computer*, November 1990.

[7] G. Krasner and S. Pope, "A description of the model-view-controller user interface paradigm in the Smalltalk-80 system," *J. Object Oriented Programming*, vol. 3, pp. 26–49, August 1988.


## 3 Biography

**Khuzaima S. Daudjee** is currently pursuing an M.Sc. degree in Computer Science at York University, Toronto, Canada. His Masters's thesis is targeted toward software reuse in toolkits and UIMSs. His interests include software engineering, user interface technology, object-oriented systems and computer-supported cooperative work. He holds a B.Sc.(Hons) degree in Computer Science from Trent University.

**Nicholas Graham** is an Assistant Professor at York University, working in the field of tools and methodologies for user interface development. Most recently, he has developed the *Clock* system [1], a rapid-prototyping language for user interfaces that supports reuse of user interface components. He has worked in language and programming environment development on the Turing Language project at the University of Toronto, as a Research Associate on the T'Nial compiler project at Queen's University, Canada, and was a visiting researcher at the German National Research Institute for Computer Science (GMD) in Karlsruhe, Germany. Graham's forthcoming Ph.D. is expected from the Technische Universität Berlin, Germany.