

SEPARATING THE USER INTERFACE FROM THE FUNCTIONALITY OF APPLICATION PROGRAMS

The following paper summarizes a presentation given at the Doctoral Consortium held at CHI '86, April 13-17, 1986.

PEDRO SZEKELY

Constructing graphical user interfaces for interactive applications is a difficult and time consuming task, typically requiring extensive programming and experimentation with many prototypes. Thus, the ability to package portions of the specification of user interfaces into components that can be reused in the construction of many interfaces, and the ability to change an application's user interface without impacting the implementation of the functionality are crucial. These abilities can be realized in the measure that the dependencies between the implementation of an application's functionality and the user interface can be minimized.

Separating the implementation of the functionality and the user interface has other benefits: multiple user interfaces can be developed for a single application, each one tailored to a different class of users, or to a different set of input and output devices; the functionality of an application can be called from another program directly, without simulating the input required by the user interface; the user interface can be specified by means other than programming, for example, by interactively drawing and demonstrating how the interface should behave.

The difficulty with separating the implementations of the functionality and the user interface is that the two parts are semantically related. An essential aspect of good user interfaces is that they have extensive knowledge about the semantics of the application they serve. Virtually all aspects of a user in-

terface, even small and seemingly unimportant ones, require knowledge about the semantics of the application.

An obvious dependency is illustrated by the specification of the display of an application data structure. The specification is part of the user interface, but the implementation requires accessing the information stored in the data structure. Thus, the user interface depends on the routines the application provides to access the data structure, or viewed the other way around, the functionality has to supply the routines the user interface needs to access the information.

Another dependency is illustrated by a common user interface feature whereby the image under the mouse cursor is highlighted to show whether it makes sense to click at it. Whether clicking at an object makes sense or not depends on application knowledge, and may also depend on the state of the application. Thus, the implementation of the user interface depends on how the application supplies this knowledge, or viewed the other way around, the implementation of the functionality depends on how the user interface requires the knowledge to be supplied.

The purpose of the thesis is to investigate ways for separating the implementation of the functionality and user interface of application programs. The thesis has two parts. The first part describes a taxonomy of the semantic dependencies between the functionality and the user interface of application pro-

grams, and also contains a study of techniques for implementing the various components of a user interface.

The taxonomy is based on a simple model of application programs that exposes the relation between the functionality and the user interface. The functionality is modeled as a set of abstract data types, and thus is viewed as a programmer's interface. The task of the user interface is first, to allow users to use the input devices to invoke the operations defined by the application's data types (input), and second, to maintain a mapping from instances of these types into a representation understood by the output devices (output). The taxonomy is done by spelling out in detail all the components involved in performing both input and output. For instance, output has three basic components: a component to translate data type instances into a representation understood by the devices, a mechanism to recognize relevant changes to the instances being presented, and a mechanism to schedule the updating of the mapping when changes are recognized.

For each component, there is a comparison of several implementation techniques with respect to separation (changes required in the implementation of the functionality if the user interface is changed), generality (ability to support the implementation of many styles of user interface) and efficiency. Many of the implementation techniques discussed are selected from some popular methodologies for implementing user interfaces (eg. Smalltalk, MacApp, subroutines libraries, Domain/Dialogue), so the analysis provides a new step towards evaluating these methodologies.

The second part of the thesis describes both a methodology for implementing user interfaces based on the results of the first part of the thesis, and a user interface construction toolkit to support the methodology. The methodology encourages the implementation of "user interface independent" programs, and the toolkit provides a set of ready made components for the implementation of graphical interfaces. An icon editor and an interface to the file system were implemented using this methodology to show the capabilities of the toolkit, and illustrate the separation of the implementation of their functionality and user interface.

The results of the thesis are of interest to application writers and designers of user interface toolkits. The taxonomy and the analysis of implementation techniques should help application writers evaluate the consequences their design decisions have on the user interface, and thus help them implement applications

with good user interfaces. The results should also help toolkit designers understand some of the issues involved in designing toolkits whose facilities can be used by a wide variety of applications.