A New Approach to Visual Programming in User Interface Design

Jürgen Herczeg, Hubertus Hohl, and Matthias Ressel
Research Group DRUID

University of Stuttgart, Department of Computer Science
Breitwiesenstr. 20-22, D-70565 Stuttgart, Germany
E-Mail: druid@informatik.uni-stuttgart.de

Keywords: Graphical user interfaces, user interface development environments, interface builders, toolkits, visual programming, direct manipulation

Abstract

To face prevailing problems with existing tools for interactively building graphical user interfaces we present a new object-oriented approach to implementing visual programming tools. This approach is employed by the user interface development environment XIT. It is based on the representation of knowledge for creating and manipulating interaction objects in the underlying user interface toolkit. This knowledge forms the basis for a set of higher-level tools, including interface builders, inspectors, browsers, and tracers, which may be applied to user interfaces created by either visual or conventional programming.

1. THE ROLE OF VISUAL PROGRAMMING IN USER INTERFACE DESIGN

Employing visual programming techniques for building graphical user interfaces is not quite new [5]. Only recently, however, with the availability of so-called *interface builders* it has attracted the attention of application programmers. With these tools, a graphical user interface may be constructed by means of direct manipulation. The user selects interface elements from palettes, composes and manipulates them with the mouse, and specifies their properties by means of menus, forms, or property sheets. In contrast to more traditional approaches to visual programming, where a *visual language* is used instead of a textual, general-purpose programming language (cf. [4]), an interface builder may be rather characterized as a "visual front end" to a special-purpose language for creating and manipulating objects provided by a *user interface toolkit*. With the help of toolkits, low-level functionality is mostly hidden from the programmer; it is encapsulated in predefined, high-level building blocks. Another major difference between visual programming in general and visual programming techniques for building graphical user interfaces is that the graphical objects manipulated by the programmer are — or at least look like — the objects to be created instead of being iconic representations of abstract concepts such as operators, symbols, or data of a programming language. So for the programmer, no transformation is needed between the program to be created and the visual representation on the screen. This closely corresponds to the very idea of direct manipulation and is probably one of the most important reasons, why visual programming turns out to be more successful in the domain of user interface design than for general programming.

2. PROBLEMS WITH EXISTING TOOLS

Although visual programming techniques seem to be most natural and predestined for building graphical user interfaces, most of the existing interface builders have several shortcomings:

- They mainly address the visual aspects of a user interface, such as display attributes and layout. Describing dynamic aspects of a user interface and connecting it to the application, which is among the most difficult tasks in building highly interactive user interfaces, is not or only partly supported.
- Many interface builders are merely add-ons to certain toolkits and, even worse, only provide parts of their already restricted functionality. Therefore, incorporating application-specific interface elements or advanced interaction techniques is not possible. Whereas general-purpose visual programming tools often have been blamed for being too low-level and not scaling up for real applications, many interface building tools can be criticized for being too high-level and therefore only suitable for standard applications.
- Interface builders cannot be used for redesigning or incrementally modifying user interfaces created by conventional programming. Therefore, modifying and reusing parts of existing programs, which is one of the key principles of an object-oriented approach, is not supported. An important aspect in user interface design, however, is investigating design alternatives and extensions for existing interfaces, rather than building new ones from scratch.
- The process of constructing a user interface with an interface builder more or less still resembles conventional programming: A graphics editor (instead of a text editor) is used to specify the overall appearance. After generating and possibly editing program code, this code has to be compiled and linked with the corresponding application program. Finally, the functionality of the interface can be tested, which usually ends up in restarting the whole procedure to perform modifications. Simulation components of interface builders for testing the behavior of a user interface are mainly restricted to application-independent dynamic aspects, such as visual feedback. Modifying or customizing interfaces at application run-time is not supported.

For some of these reasons, today, most user interfaces are still implemented by means of conventional programming with the help of user interface toolkits or even no tools at all [7]. Visual programming techniques, however, can be useful for many different aspects of user interface design. Assembling user interface elements and inspecting and manipulating their (visual) properties is the basic functionality common to all interface builders. In addition, more advanced techniques have been developed for various, mostly research tools. Among them are (1) the creation of new interface elements from low-level graphical elements, as for example in Peridot [5] and Garnet [6]; (2) establishing links among user interface elements or between the user interface and the application, e.g., by direct manipulation in the NeXT Interface Builder [8]; (3) defining dependencies between them, e.g., by interactively setting up constraints in Garnet; and (4) specifying the dynamic behavior of a user interface, e.g., by demonstration in Peridot and DEMO [9].

3. A NEW APPROACH TO VISUAL PROGRAMMING

In order to face most of the problems described above we have used an approach which can basically be characterized as follows:

1. More functionality is integrated into the lower-level tools. Instead of providing a more or less complete, but rather static set of widgets, we have developed a powerful and “knowledgeable”

user interface toolkit, which provides a rich programming interface and subsumes most of the functionality needed for the higher-level tools.

2. The functionality of the interface builder is split up and distributed among a set of smaller, more universal interactive tools mutually invoking each other. They are based on the fundamental properties of the toolkit and therefore are also applicable to user interfaces created by conventional programming.
3. There is no separation between the development environment and the application environment. Therefore, each of the interactive tools may be equally used at application run-time.

In simplified terms, the key principle of this approach is: Instead of letting the interface builder know how a particular interface element may be created and manipulated, the interface element itself should know.

3.1. The User Interface Development Environment XIT

This approach has been implemented by the user interface development environment XIT, which is based on the Common Lisp Object System (CLOS) and the X Window System. XIT includes the following tools:

- *User interface toolkit* providing an object-oriented programming interface
- *User interface metasystem* for interactively inspecting and manipulating user interfaces
- *User interface construction kit* for composing interaction objects by direct manipulation
- *User interface resource editor* for adjusting resource attributes to application-specific or user-specific preferences
- *User interface browser* for inspecting the structural dependencies of a user interface and its underlying application
- *User interface tracer* for inspecting the behavioral dependencies of a user interface in a running application

Some of these tools have already been described in [2]. In the following sections, we will especially concentrate on the functionality which makes them different from related tools.

3.2. User Interface Toolkit

To provide an easy-to-use, but also general and extensible programming interface, XIT is based on two layers of user interface toolkits: (1) A *low-level toolkit*, which provides a general framework for building all kinds of interaction objects, and (2) a *high-level toolkit* providing an extensible set of common interaction objects, such as buttons, menus, property-sheets, etc., which are instantiations of the low-level toolkit.

In addition to standard and advanced features of these toolkits, which are described in [1], interaction objects in XIT have knowledge about (1) how they may be created or copied, (2) what characteristic attributes they have and how these may be visualized and manipulated, (3) what operations may be performed on them, and (4) how program code may be created for them. This knowledge is represented in the corresponding interaction objects classes, which are organized into an inheritance hierarchy. Therefore, new interaction objects generated by subclassing or aggregation automatically obtain this knowledge. Additional knowledge, e.g., for application-specific interaction objects, may be easily added if required.

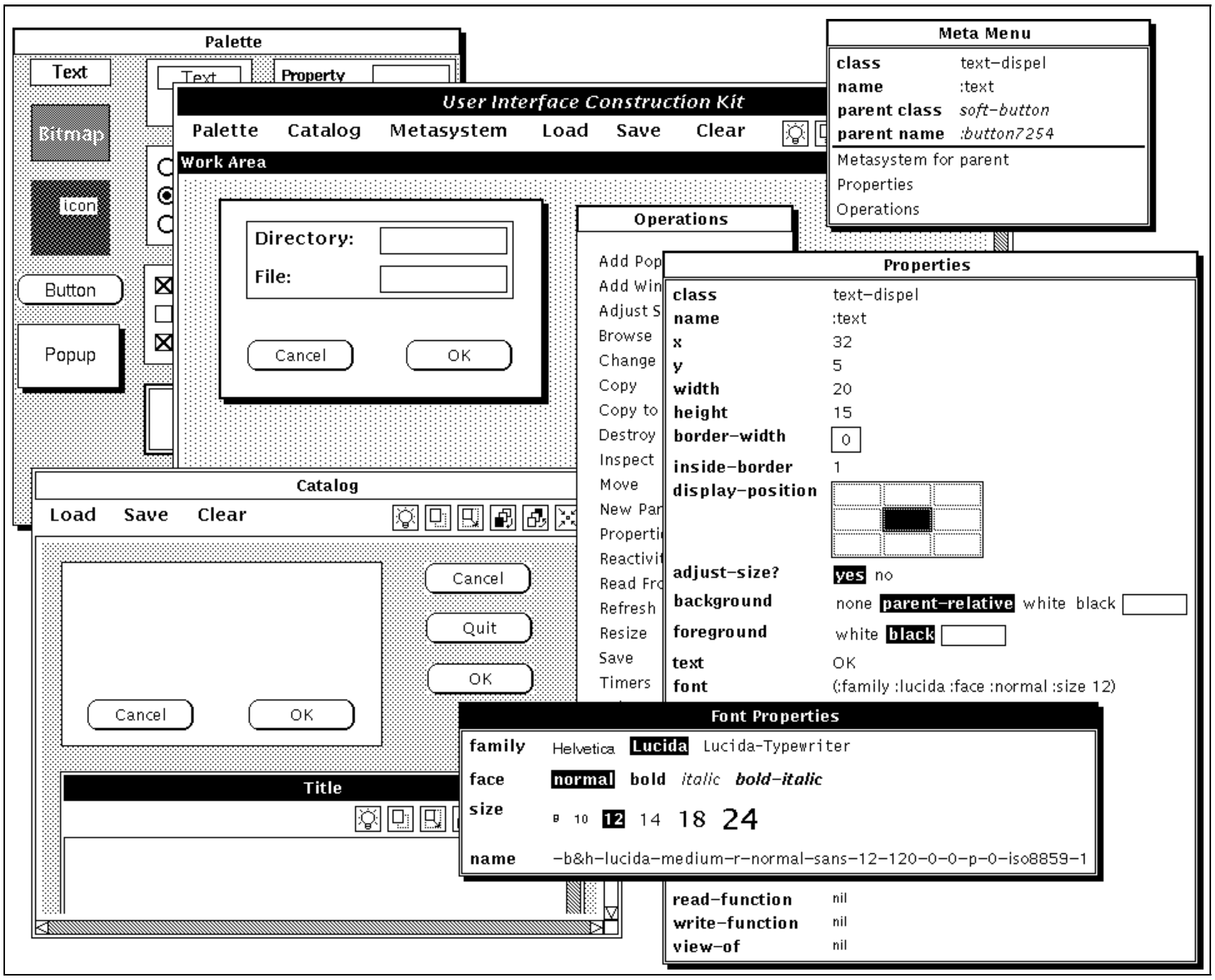

Figure 1: User Interface Construction Kit and Metasystem

3.3. User Interface Metasystem and Construction Kit

The *user interface metasystem* provides a graphical interface for inspecting and manipulating *properties* of interaction objects and performing *operations* on them. It may be invoked for any kind of interaction object, no matter whether it has been created by visual or conventional programming, and even in a running application. Properties, which may be either visual attributes, e.g., regarding the geometry of an interaction object, or behavioral attributes, e.g., describing the mapping of events onto the corresponding actions, are presented in property sheets. Attribute values are presented by editable text fields, menus, sliders, or separate property sheets, e.g., for font or color attributes. Operations, such as moving, resizing or copying an interaction object, are provided by menus. All property sheets and menus are object-specific; their contents is automatically generated from the knowledge represented for the object in the toolkit.

The *user interface construction kit* makes use of the metasystem and additionally provides common interaction objects in the form of *palettes*. Different palettes contain different kinds of objects. *Catalogs* provide user-extensible sets of (more complex) interaction objects that may be saved and reloaded. A catalog contains “real” interaction objects rather than iconic representations; selecting one of them simply creates a copy. Saving and reloading interaction objects is a basic functionality provided by the toolkits, which is used by the construction kit to store and retrieve an

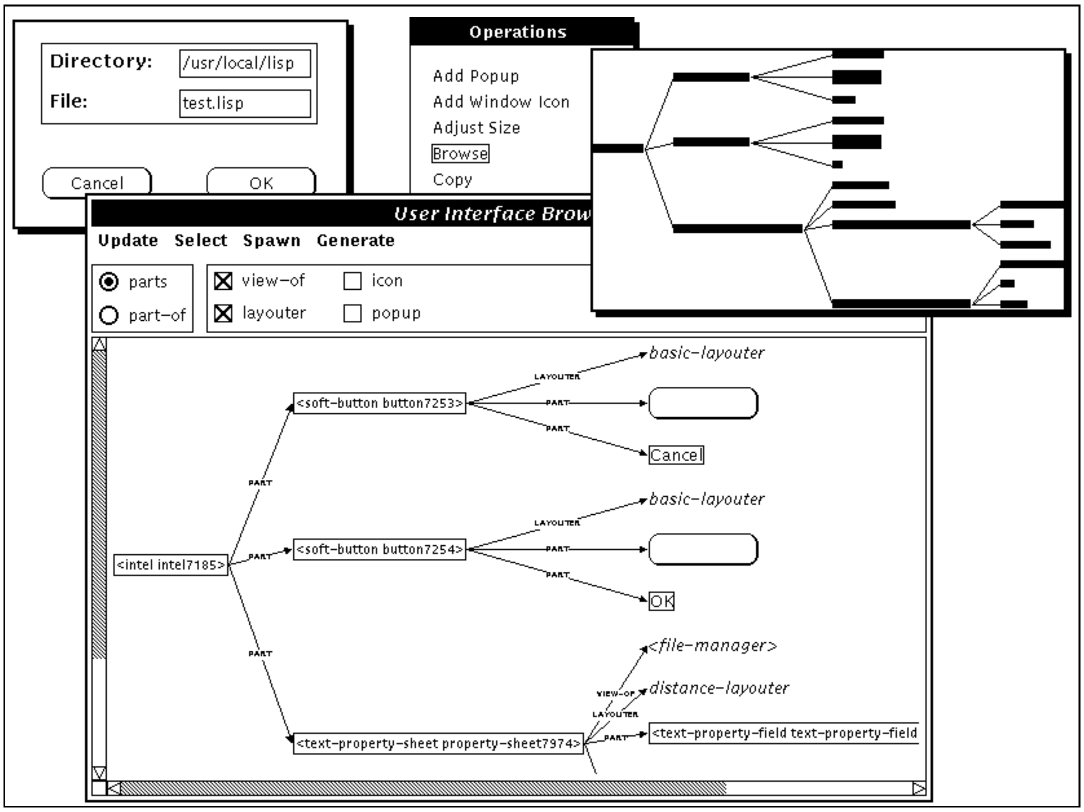

Figure 2: User Interface Browser

interface built by the user. Objects are stored in an executable, textual form, which may be edited if required and directly loaded by an application. Manipulating interfaces built by means of the construction kit and connecting them to the application is performed with the metasystem. Figure 1 shows different components of the construction kit and the metasystem building a dialog window for specifying a file.

By creating application-specific palettes, catalogs and metasystem components, the user interface construction kit may serve as an application builder.

3.4. User Interface Browser and Tracer

Interaction objects are organized into object networks. Each object is part of an interaction object hierarchy. Attribute values may be complex objects themselves, e.g., objects describing the layout or behavior. Finally, interaction objects are connected to application objects, which in turn may be viewed by an arbitrary number of interaction objects. The user interface metasystem only indirectly reflects these interdependencies by allowing the user to switch to parent or subpart objects and by providing special property sheets for complex attribute values. However, the overall functionality of an interaction objects can be made more visual and intelligible by explicitly presenting these dependencies in a graph. This is performed by the *user interface browser* which can be invoked by the “Browse” operation of the metasystem. Figure 2 shows the browser displaying the interaction object hierarchy for the dialog window created in figure 1. The dependencies to be visualized can be selected from menus. The graphical output displayed in the browser may not only be used for viewing but also for manipulating dependencies. Also for each interaction object presented in the graph, the metasystem may be invoked to inspect and modify properties.

The user interface browser as well as the metasystem basically display static aspects of a user interface. Dynamic aspects, i.e., which operations are performed in reaction to which events, are only visualized in the form of static event-action mappings in the metasystem. Understanding the internal dynamic behavior of an interface or locating errors can be very cumbersome or even impossible. For this reason, we are currently implementing a *user interface tracer* which can be used to visualize actions invoked in response to events both textually and graphically. The user may specify which actions for which events should be traced. Object-specific traces can be specified by means of the metasystem. The user interface browser and tracer are based on a general framework for building browsing and tracing tools in object-oriented programming environments [3].

4. LESSONS LEARNED

By representing knowledge about building, inspecting, and manipulating graphical user interfaces in the lower-level components, i.e. the toolkits, of the user interface development environment XIT, the implementation of visual programming tools was quite straightforward and consistent. Since these tools can be applied to any user interface built from the XIT toolkits, by either visual or conventional programming and even at application run-time, they not only qualify for user interface designers and implementors, but also for end users. Enhancing the knowledge of dealing with interaction objects by higher-level knowledge of user interface design, e.g., concerning human aspects, could substantially add to the usability of these tools.

REFERENCES

[1] J. Herczeg, H. Hohl, and M. Ressel. Progress in Building User Interface Toolkits: The World According to XIT. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, November 1992.

[2] J. Herczeg, H. Hohl, and T. Schwab. XIT - A Multi-Layered Tool for User Interface Design. In *Human Aspects in Computing, Proceedings of the Fourth International Conference on Human-Computer Interaction*, volume 1, pages 678–683, Stuttgart, September 1991.

[3] H. Hohl, J. Herczeg, and M. Ressel. An Interactive Design Environment for Graphical Browsers. This volume.

[4] B. A. Myers. Visual Programming, Programming by Example and Program Visualization: A Taxonomy. In *CHI'86 Proceedings*, pages 59–66. ACM, April 1986.

[5] B. A. Myers and W. Buxton. Creating Highly-Interactive and Graphical User Interfaces by Demonstration. *Computer Graphics, Proceedings of SIGGRAPH 86*, pages 249–258, August 1986.

[6] B. A. Myers et al. Garnet – Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[7] B. A. Myers and M. B. Rossen. Survey on User Interface Programming. In *CHI-92 Conference Proceedings, Human Factors in Computing Systems*, pages 195–202. ACM SIGCHI, 1992.

[8] B. F. Webster. *The NeXT Book*. Addison-Wesley Publishing Company, 1989.

[9] D. Wolber and G. Fisher. A Demonstrational Technique for Developing Interfaces with Dynamically Created Objects. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 221–230, November 1991.