Visuelle Programmierung graphischer Benutzungsoberflächen: Ein wissensbasierter, generischer Ansatz

Jürgen Herczeg
Universität Stuttgart

Zusammenfassung

Durch die Verbreitung graphischer Benutzungsoberflächen-Editoren setzt sich visuelle Programmierung bei der Entwicklung von Benutzungsschnittstellen immer mehr durch. Die meisten der verfügbaren Werkzeuge schöpfen jedoch das Potential interaktiver visueller Programmierung nur unzureichend aus und besitzen in bezug auf Funktionsumfang und Flexibilität zahlreiche Schwächen. Diese liegen oftmals bereits in den verwendeten Toolkits begründet. Der vorliegende Beitrag zeigt Probleme bestehender Werkzeuge auf. Es wird ein Ansatz vorgestellt, mit dem visuelle Programmierwerkzeuge durch Repräsentation von Wissen auf Toolkit-Ebene teilweise automatisch generiert werden können. Diese bilden ein „Metasystem“, mit dem Benutzungsschnittstellen auch zur Anwendungslaufzeit erstellt und modifiziert werden können, und ermöglichen so eine sinnvolle Kombination konventioneller und visueller Programmierung. Die generische Modellierung der Werkzeuge gewährleistet einfache Erweiterbarkeit. Darüberhinaus ermöglicht der gewählte wissensbasierte Ansatz auch die Unterstützung des Benutzers, beispielsweise bei der Berücksichtigung ergonomischer Entwurfsrichtlinien.

1 Visuelle Programmierung graphischer Benutzungsoberflächen

Die Möglichkeit, graphische Benutzungsoberflächen durch *visuelle Programmierung* zu erstellen, wurde bereits mit großem Interesse verfolgt, bevor diese einem größerem Anwenderkreis zugänglich waren. PERIDOT [10] war ein Vorläufer solcher *visuellen Programmierwerkzeuge*. Durch die Verbreitung graphischer Benutzungsoberflächen und den steigenden Bedarf an effizienten Entwicklungsmethoden und -werkzeugen haben sich graphische *Benutzungsoberflächen-Editoren* (*Interface Builder*) als Bestandteile von *User Interface Management Systemen (UIMS)* mittlerweile etabliert; Beispiele dafür sind der NEXT Interface Builder [19] und DEVGUIDE [16]. Diese erlauben es, Teile einer Benutzungsschnittstelle graphisch interaktiv ohne den direkten Einsatz einer konventionellen Programmiersprache zu erstellen. Sie arbeiten üblicherweise nach dem Baukastenprinzip unter Verwendung direkter Manipulation. Wesentliche Unterschiede bestehen allerdings darin,

- welche Aspekte *visuell* erstellt werden können: statisches Layout, Dynamik, Anwendungsanbindung;
- welche Interaktionstechniken zur Verfügung stehen: nur vordefinierte „Standard-Bausteine" oder beliebige durch den Benutzer definierbare *Interaktionsobjekte*;
- welche Implementierungsplattformen unterstützt werden: Programmiersprachen, Fenstersysteme, Baukästen (*Toolkits*).

Kommerzielle Werkzeuge konzentrieren sich im Hinblick auf „Standard-Anwendungen" mehr und mehr auf letzteren Aspekt mit dem Ziel plattformunabhängiger *portabler Benutzungsoberflächen*. Mit der Bindung an vorgefertigte Bausteine und strenge Entwurfsrichtlinien (*User Interface Style-Guides*) schränken sie jedoch notwendigerweise das Spektrum erstellbarer Benutzungsoberflächen – und damit auch die Kreativität des Entwicklers – stark ein. Systeme, die im Bereich der Forschung entwickelt und eingesetzt werden, wie z. B. GARNET [13] oder die im folgenden vorgestellten Entwicklungsumgebung XIT, decken ein wesentlich breiteres Spektrum von Anwendungen ab (z. B. direkte Manipulation oder interaktive Graphik) sowie Entwurfsmöglichkeiten, die deutlich über statisches Layout hinausgehen.

1.1 Schwächen visueller Programmierwerkzeuge

Visuelle Programmierung bietet sich auf natürliche Weise an, wenn die zu erstellenden Konstrukte visueller Art sind. Der Anwendung des WYSIWYG-Paradigmas ist es zuzuschreiben, daß visuelle Programmierung bei graphischen Benutzungsschnittstellen wesentlich erfolgreicher ist als bei interner Anwendungslogik. Dennoch werden Benutzungsschnittstellen trotz einer Vielzahl verfügbarer Werkzeuge größtenteils *nicht* durch visuelle, sondern durch konventionelle Programmierung – meist unter Einsatz von Baukästen (*Toolkits*) – erstellt [12]. Verschiedene Gründe sind dafür verantwortlich:

- Graphische Editoren beschränken sich auf die Bearbeitung der visuellen Aspekte einer Benutzungsschnittstelle, wie etwa Darstellungsattribute (Farben, Fonts, etc.) und Layout. Die Beschreibung der Dynamik und die Anbindung an die Anwendung – die weitaus schwierigeren Aufgaben – werden nur selten graphisch unterstützt.

Durch *demonstratives Programmieren*, das in Systemen wie DRUID [14] oder MARQUISE [11] eingesetzt wird, ist es lediglich möglich, graphisch visualisierte, nicht jedoch interne dynamische Aspekte zu beschreiben.

- Zur Spezifikation der Dynamik werden werkzeugspezifischen Dialogbeschreibungssprachen oder konventionellen Programmiersprachen verwendet, die meist regelbasiert oder prozedural sind. Dies führt zu einem Bruch mit dem objektbasierten visuellen Programmierparadigma.

- Viele Unzulänglichkeiten graphischer Werkzeuge, wie beispielsweise unzureichende Mechanismen zur Anbindung an die Anwendung, liegen bereits in den verwendeten Toolkits begründet [4]. Der Mangel an Ausdrucksmöglichkeiten wird durch die visuelle Programmierschnittstelle zwangsläufig noch verstärkt. So wird etwa die auf Toolkit-Ebene angebotene Möglichkeit zur Definition neuer Interaktionsobjekte graphisch nicht unterstützt.

- Die meisten Interface Builder sind bezüglich ihrer Benutzungsoberfläche und dem Funktionsumfang statisch und können nicht erweitert werden. Das Spektrum wünschenswerter Erweiterungen reicht von zusätzlichen Interaktionsobjekten bis hin zu anwendungsspezifischer Funktionalität.

- Interaktive Anwendungen werden nach wie vor durch traditionelle Stapelverarbeitung erstellt: Die Benutzungsoberfläche wird mit einem Graphikeditor (statt einem Texteditor) konstruiert, übersetzt und an die Anwendung angebunden. Erst jetzt entsteht ein lauffähiges Programm, mit dem Oberfläche und Anwendung zusammen getestet werden können. Notwendige Modifikationen zwingen dazu, viele solcher Zyklen zu durchlaufen. Simulationskomponenten, die oft zusätzlich angeboten werden, erlauben es lediglich, dynamisches Verhalten auf syntaktischer Ebene zu testen. Semantische Rückmeldungen, wie sie für direkt-manipulative Anwendungen charakteristisch sind, lassen sich nicht einbeziehen.

- Als Ausgabe wird Programmcode in einer werkzeugspezifischen oder allgemeinen Programmiersprache erzeugt. Umgekehrt können jedoch Benutzungsschnittstellen, die durch konventionelle Programmierung erstellt wurden, durch visuelle Werkzeuge *nicht* nachbearbeitet oder eingebunden werden. Die Wiederverwendbarkeit vorhandener Module wird somit trotz eines objektorientierten Ansatzes nicht unterstützt.

Der inkrementellen und explorativen Vorgehensweise, die für den Entwurf graphischer Benutzungsschnittstellen typisch ist, werden bestehenden Entwicklungswerkzeuge aufgrund dieser Schwächen nicht gerecht. Das Potential interaktiver visueller Programmierung wird nur ansatzweise ausgeschöpft. Interaktive objektorientierte Programmierumgebungen, wie beispielsweise SMALLTALK-80, aus denen sich Benutzungsschnittstellen-Entwicklungsumgebungen ursprünglich entwickelt haben, zeichnen sich durch Werkzeuge aus, mit denen jeder Programmaspekt – insbesondere auch zur Laufzeit – inspiziert und manipuliert werden kann. Bei graphischen Benutzungsoberflächen ist dies aufgrund der strikten Trennung zwischen Entwicklungs- und Laufzeitumgebung nur in eingeschränktem Maße möglich: *Window-Manager* ermöglichen es dem Benutzer, Anwendungsfenster, z. B. bezüglich ihrer Position und Größe, zu manipulieren. Mit *Ressource-Editoren* können Attribute wie Farben oder Schriftarten geändert werden. Ebenso einfach sollte es beispielsweise für den Benutzungsschnittstellen-Entwickler oder auch Endbenutzer sein, interaktiv die Einträge eines Anwendungsmenüs umzuordnen, neu zu benennen oder weitere Einträge hinzuzufügen.

1.2 Ein wissensbasierter, generischer Ansatz

In *wissens-* oder *modellbasierten* Systemen, wie z. B. UIDE [2], wird die Benutzungsschnittstelle automatisch aus der Anwendung und einer Beschreibung ihrer Semantik generiert. Das Anwendungsmodell spiegelt sich dabei unmittelbar auf der Benutzungsoberfläche wider. Da das Hauptinteresse auf der Anwendungslogik liegt, entstehen jedoch in aller Regel wenig attraktive Oberflächen. Dies kann durch Einbeziehen visueller Programmiertechniken, wie etwa in HUMANOID [18], verbessert werden.

Im folgenden wird ein wissensbasierter Ansatz vorgestellt, bei dem wesentliche Teile der Benutzungsschnittstelle visueller Programmierwerkzeuge automatisch generiert werden. Das dazu benötigte Wissen wird nicht wie in traditionellen wissensbasierten Systemen explizit in den Werkzeugen, sondern bei den Interaktionsobjekten des Toolkits selbst repräsentiert. Die Werkzeuge liegen bei diesem generischen Ansatz nur als Rahmensysteme vor und werden durch die bearbeiteten Objekte spezialisiert; d. h. nicht der Editor „weiß“, wie Benutzungsschnittstellen erstellt und manipuliert werden, sondern die darin enthaltenen Bausteine. Daraus ergeben sich zahlreiche Vorteile [5]:

- Das dem Toolkit zugrunde liegende Modell wird von den visuellen Werkzeugen vollständig wiedergegeben; diese verfügen nahezu über die gesamte interne Funktionalität bezüglich Präsentation, Dynamik und Anwendungsschnittstelle. Modifikationen und Erweiterungen des internen Modells, z. B. neue Interaktionsobjekte, stehen automatisch auch auf der Benutzungsoberfläche zur Verfügung.

- Benutzungsschnittstellen können nicht nur aus den angebotenen Bausteinen konstruiert werden, sondern es lassen sich auch Oberflächen visuell bearbeiten, die auf Toolkit-Ebene programmiert wurden. Visuelle und konventionelle Programmierung können sich also sinnvoll ergänzen.

- Benutzungsschnittstellen können auch zur Programmlaufzeit bearbeitet werden; eine Simulationskomponente wird somit nicht benötigt. Es besteht keine strikte Trennung mehr zwischen Entwicklungs- und Laufzeitumgebung. Daraus ergibt sich auch die Möglichkeit zum Modifizieren oder Individualisieren einer Oberfläche durch den Endbenutzer.

Durch den wissensbasierten Ansatz werden die Werkzeuge zu *Metasystemen* für graphische Benutzungsschnittstellen, sogenannten *Meta-Benutzungsschnittstellen* [6]. Einfache Beispiele dafür wurden bereits in USIT [7] und INTERVIEWS [9] implementiert. Diese waren jedoch statisch und auf allgemeine Eigenschaften von Interaktionsobjekten (z. B. geometrische Attribute) beschränkt. Im folgenden wird ein Metasystem beschrieben, das auch objektspezifische Eigenschaften berücksichtigt.

2 Ein Metasystem für Interaktionsobjekte

Ein generisches Metasystems für Interaktionsobjekte wurde als Bestandteil der Benutzungsschnittstellen-Entwicklungsumgebung XIT realisiert.

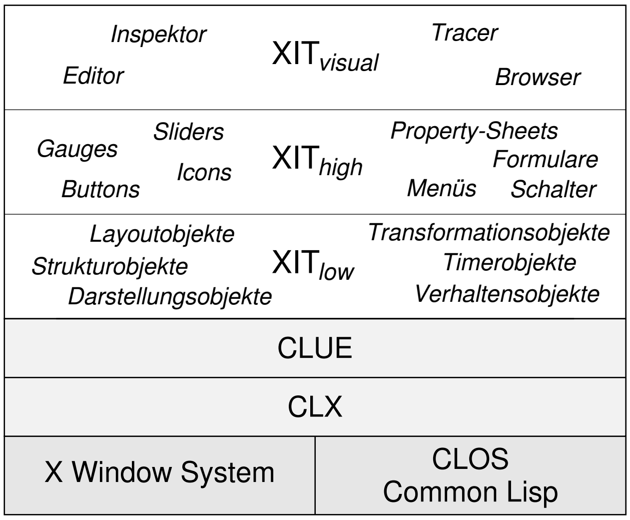

Abb. 1: Systemarchitektur von XIT

2.1 Systemarchitektur

Abb. 1 zeigt die geschichtete Systemarchitektur von XIT[3]. XIT ist in COMMON LISP und CLOS implementiert und benutzt CLX und CLUE [8] als Schnittstellen zum X Window System. Darauf bauen zwei Toolkit-Ebenen auf: XIT$_{low}$, ein Rahmensystem für Interaktionsobjekte, und XIT$_{high}$ eine erweiterbare Interaktionsobjekt-Bibliothek. Diese objektorientierten Baukästen bilden die Wissensbasis für die visuellen Programmierwerkzeuge des Metasystems XIT$_{visual}$. Der Editor, Inspektor und Browser aus XIT$_{visual}$ werden im folgenden näher beschrieben.

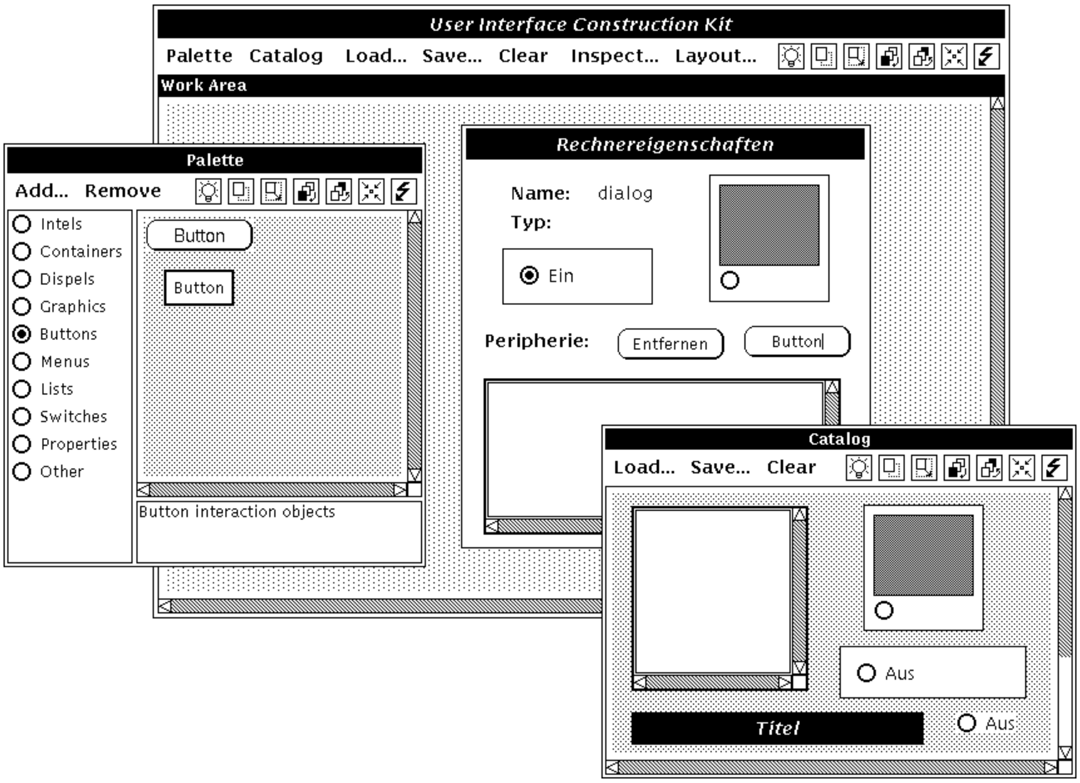

Abb. 2: Interaktionsobjekt-Editor

2.2 Interaktionsobjekt-Editor

Zum Erzeugen und Verknüpfen von Interaktionsobjekten kommt die in graphischen Editoren verbreitete Baukasten-Metapher zur Anwendung. Dabei werden Objekte durch Drag&Drop aus einer *Palette* in den *Arbeitsbereich* gezogen. Abb. 2 zeigt beispielsweise, wie mit dem Interaktionsobjekt-Editor ein Fenster zur Darstellung der Eigenschaften eines Rechners erzeugt wird.

Die verfügbaren Interaktionsobjekte sind über verschiedene Paletten verteilt. Es existieren Paletten für die Standard-Bausteine von XIT$_{high}$(Buttons, Menüs, Property-Sheets etc.) sowie für die in XIT$_{low}$ enthaltenen Grundbausteine (*Darstellungsobjekte* für Text, Bild und Graphik, allgemeine *Strukturobjekte*); aus diesen können neue Interaktionsobjekte zusammengesetzt werden, wie etwa das mit einem Ein/Aus-Schalter versehene graphische Bildschirmobjekt (s. Abb. 2). Die Verteilung der Objekte über verschiedene Paletten ermöglicht es, eine große Anzahl von Bausteinen anzubieten, diese semantisch zu gruppieren und bei Bedarf auch Paletten mit anwendungsspezifischen Bausteinen nachzuladen. Abb. 3 zeigt beispielsweise, wie über eine Palette mit speziellen Icons Rechner und Peripheriegeräte graphisch konfiguriert werden können.

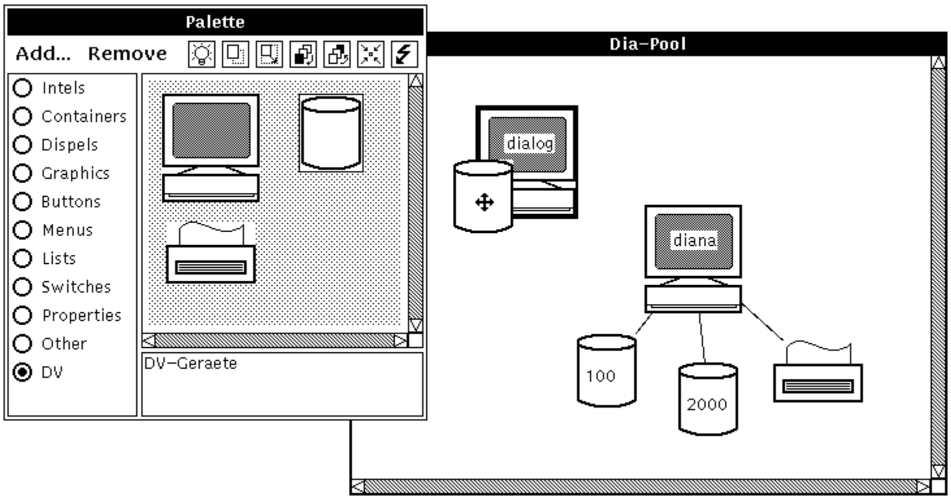

Abb. 3: Anwendungsspezifische Palette zur Konfiguration von Rechnerobjekten

Paletten werden aus internen *Palettenbeschreibungen* generiert. Diese enthalten für jeden Eintrag die zugehörige graphische Darstellung und die Aktion, die bei dessen Auswahl ausgelöst wird. Im einfachsten Fall wird lediglich auf eine Interaktionsobjekt-Klasse verwiesen, die an der jeweiligen Zielposition instantiiert wird. Allgemein können Paletteneinträge jedoch beliebige Aktionen auslösen. So werden etwa durch die Bausteine der Palette in Abb. 3 zusätzlich zu Interaktionsobjekten auch entsprechende Anwendungsobjekte erzeugt und verknüpft. Der Benutzungsoberflächen-Editor wird auf diese Weise zum Anwendungseditor.

Bei der Auswahl eines Paletteneintrags wird kein graphisches Abbild, sondern das tatsächliche Interaktionsobjekt instantiiert; dieses kann deshalb nicht nur im Arbeitsbereich des Editors, sondern in einem beliebigen Anwendungsfenster abgelegt werden (vgl. Abb. 3). Der gesamte Bildschirm wird somit zum Arbeitsfläche. Die erzeugten Objekte sind voll funktionsfähig; es wird nicht zwischen einem Konstruktions- und einem Testmodus unterschieden. Auf diese Weise können mit dem Editor Anwendungen auch zur Laufzeit – beispielsweise um zusätzliche Buttons oder Menüeinträge – erweitert werden.

Durch Drag&Drop-Operationen können Objekte – abhängig von Quellobjekt, Zielobjekt und Zielposition – (a) geometrisch *positioniert*, (b) *aggregiert* (d. h. ineinander verschachtelt) oder (c) miteinander *assoziiert* werden. In Abb. 3 werden durch Ziehen des Festplatten-Icons aus der Palette auf ein Rechner-Icon eine Kombination dieser verschiedenen Aktionen ausgeführt: Das entsprechende Icon wird (1) instantiiert, (2) in das umgebende Fenster (in einem vorgegeben Abstand vom Rechner-Icon) eingefügt und (3) mit diesem durch eine Linie verbunden; (4) die zugrundeliegenden Anwendungsobjekte werden miteinander assoziiert. Das Wissen darüber, welche Aktionen durch Drag&Drop ausgelöst werden, ist bei den beteiligten Objekten repräsentiert; z. B. „weiß" ein Platten-Icon, daß es nur auf einem Rechner-Icon abgelegt werden darf; das betreffende Rechner-Icon führt die Verknüpfung der zugeordneten Anwendungsobjekte durch.

Einfacher als das Erzeugen von Interaktionsobjekten aus Bausteinen der Paletten ist oftmals das Kopieren bereits vorhandener, ähnlicher Objekte. Beispielsweise kann ein neuer Menüeintrag durch Kopieren eines bestehenden Eintrags und Modifizieren von Eigenschaften der Kopie (z. B. Text und Aktion) erzeugt werden. Eine generische Kopieroperation erlaubt es, auch komplexe Interaktionsobjekte (z. B. komplette Anwendungsfenster) zu kopieren. In der Objektkopie werden auch die entsprechenden Verweise auf Anwendungsobjekte hergestellt. Das zur Implementierung der Kopieroperation notwendige Wissen ist bei den Interaktionsobjekten definiert. Nach demselben generischen Prinzip arbeitet auch die Code-Generierung, die zum Abspeichern der Objekte im Arbeitsbereich dient. Sie ist deshalb nicht auf die in den Paletten enthaltenen Objekte beschränkt, sondern erzeugt für *jedes* XIT-Interaktionsobjekt den zugehörigen Lisp-Programmcode.

Der *Katalog* (vgl. Abb. 2) ist eine durch den Benutzer interaktiv erweiterbare Palette für komplexere Bausteinen. Er kann als Zwischenspeicher für Copy&Paste-Operationen dienen oder – durch die Möglichkeit, den Inhalt abzuspeichern und zu laden – zum Austausch von Bausteinen über verschiedene Anwendungen hinweg. Das Einfügen und Entnehmen von Objekten ist über die generische Kopieroperation realisiert.

2.3 Interaktionsobjekt-Inspektor und -Browser

Da die Bausteine in herkömmlichen Benutzungsoberflächen-Editoren fest vorgegeben sind, genügen zur Manipulation ihrer Attribute vordefinierte, statische Property-Sheets. Diese sind nur auf die im Arbeitsbereich des Editors enthaltenen Objekte anwendbar, nicht aber auf die entsprechenden Objekte in der späteren Anwendung. In XIT wurde dazu der *Interaktionsobjekt-Inspektor* entwickelt. Er ermöglicht das Inspizieren und Manipulieren der Eigenschaften jedes beliebigen Interaktionsobjektes und den Einstieg in andere Werkzeuge des Metasystems.

Der Aufruf des Inspektors erfolgt durch Betätigen des *Metasystem-Buttons* (`Select Me-`

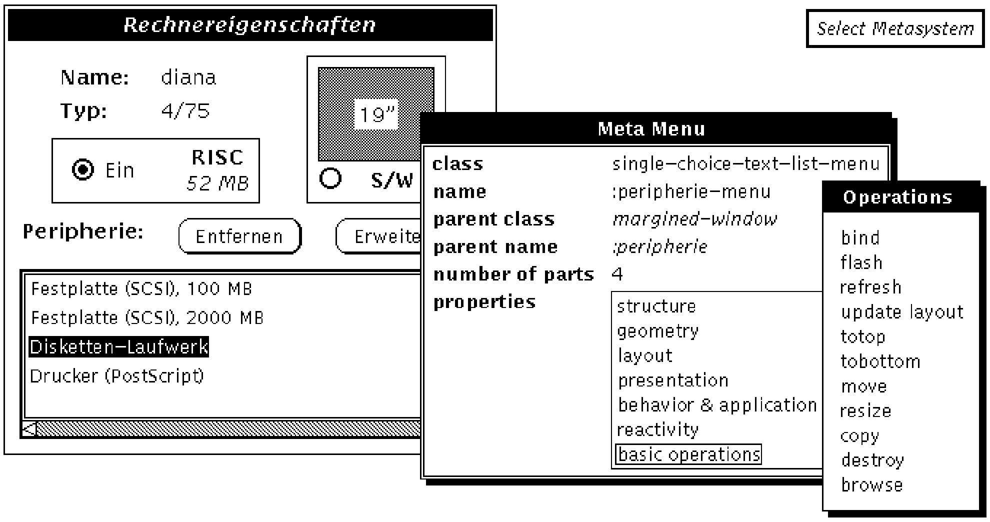

Abb. 4: Aufruf des Metasystems

`tasystem`) und anschließende Auswahl des gewünschten Interaktionsobjekts, in Abb. 4 beispielsweise die Peripherieliste des erstellten Eigenschaftsfensters; für Objekte, die sich im Arbeitsbereich des Editors befinden, genügt ein spezieller Mausklick. Daraufhin öffnet sich das *Metamenü*, das Auskunft über Name und Klasse des betreffenden Interaktionsobjekts sowie über hierarchisch über- und untergeordnete Objekte gibt. Es werden verschiedene Eigenschaftskategorien zur Auswahl gestellt, zu denen spezielle Inspektoren individuelle Objektsichten bieten. Über den Eintrag `basic operations` wird ein Menü geöffnet, das u. a. Operationen zum Bewegen, Kopieren oder Löschen des Objekts anbietet (s. Abb. 4).

Wird eine Eigenschaftskategorie aus dem Metamenü ausgewählt, so erscheint ein Property-Sheet, das die zu dieser Kategorie gehörigen Objektattribute und -operationen enthält. Abb. 5 zeigt beispielsweise die Property-Sheets für Struktur-, Geometrie- und Layouteigenschaften. Jedes Attribut wird durch einen attributspezifischen Eintrag visualisiert, der den aktuellen Wert des inspizierten Objekts darstellt und geeignete Interaktionen zu dessen Manipulation erlaubt. Handelt es sich bei den Attributwerten selbst um komplexere Objekte, wie etwa die zur geometrischen Anordnung von Interaktionsobjekten verwendeten *Layoutobjekte* (vgl. Abb. 5), oder steht eine große Zahl möglicher Attributwerte zur Auswahl, so werden zu deren Visualisierung eigene Property-Sheets und Paletten verwendet. Dies trifft auch auf die zur Beschreibung von Präsentationseigenschaften verwendeten Darstellungs-Ressourcen zu, wie z. B. Farben, Fonts, Bitmaps und Cursor (vgl. Abb. 6). Alle Property-Sheets werden beim Aufruf mit den für das betreffende Objekt spezifischen Einträgen dynamisch generiert und aus Effizienzgründen zur Wiederverwendung zwischengespeichert.

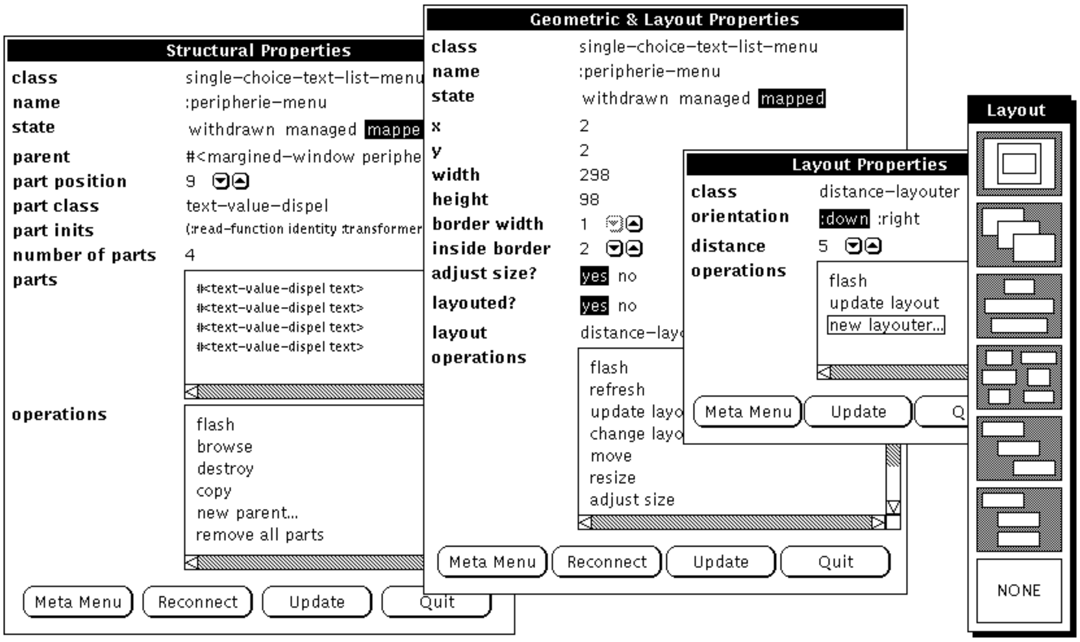

Abb. 5: Property-Sheets für Struktur-, Geometrie- und Layouteigenschaften

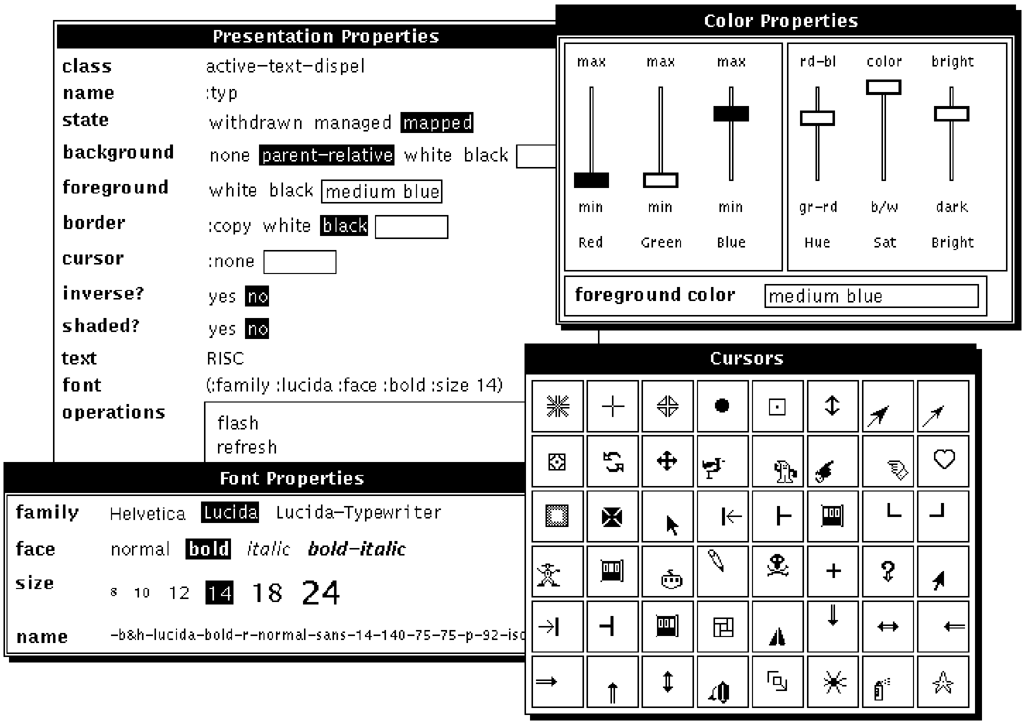

Abb. 6: Property-Sheets und Paletten für Präsentationseigenschaften

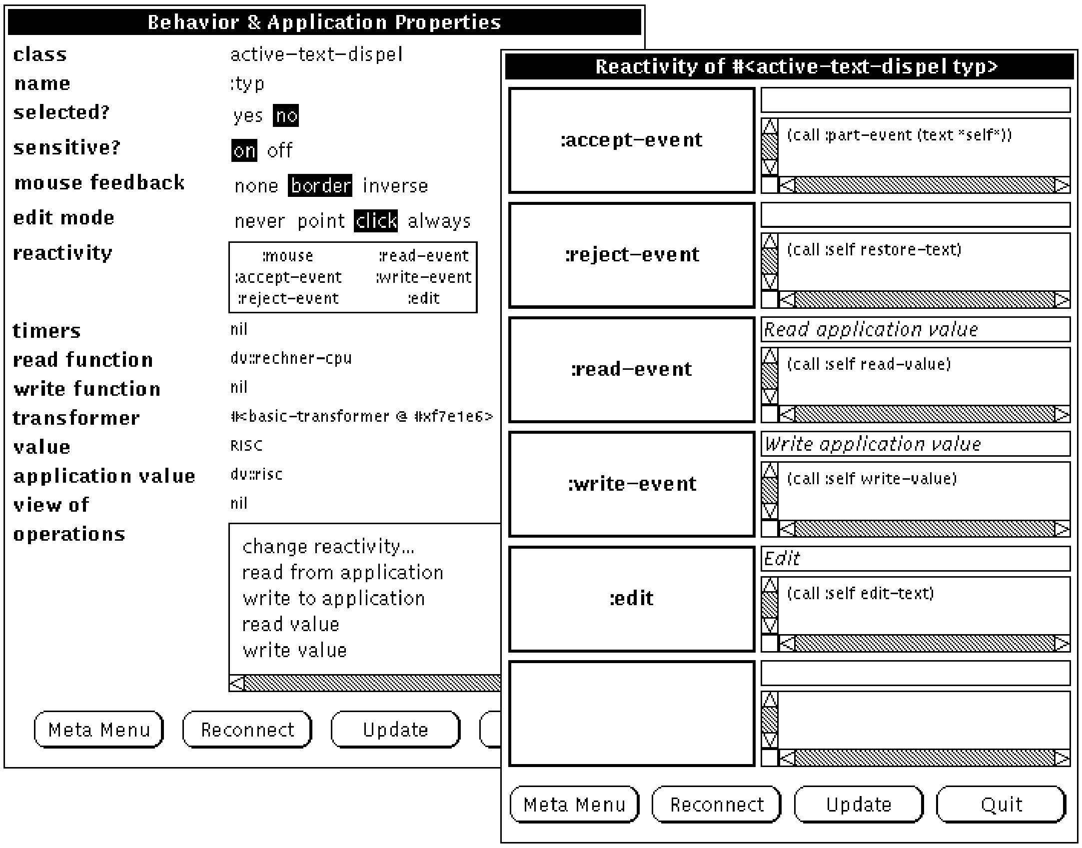

Abb. 7: Property-Sheets für dynamisches Verhalten und Anwendungsanbindung

Jedes Property-Sheet kann über den `Reconnect`-Button an andere Objekte desselben Typs gebunden werden. Wird dieser gedrückt, so sind alle Bildschirmobjekte dieses Typs sensitiv. Sollen verschiedene Objekte gleiche Eigenschaften erhalten, so ist es einfacher, anstatt dieselben Attributwerte mehrfach einzugeben, diese von einem Objekt zum anderen zu kopieren. Das Kopieren erfolgt aus dem betreffenden Attribut-Eintrag im Property-Sheet durch Zeigen auf das Objekt, von dem der Wert übernommen werden soll; dabei sind diejenigen Bildschirmobjekte sensitiv, die über das entsprechende Attribut verfügen. Kopiert werden können Präsentationsattribute (um Objekten gleiches Aussehen zu geben), geometrische Attribute (um Objekte einheitlich anzuordnen), Strukturattribute (um beispielsweise alle Einträge eines Menüs in ein anderes zu kopieren) oder auch Attribute, die das dynamische Verhalten eines Objekts beschreiben.

Da in XIT dynamisches Verhalten und Anwendungsanbindung ebenfalls über Objektattribute spezifiziert werden, können auch diese mit dem Inspektor bearbeitet werden. Abb. 7 zeigt die entsprechenden Property-Sheets für ein editierbares Textfeld. Verweise auf Anwendungsobjekte und -operationen können durch Texteingabe interaktiv einge-

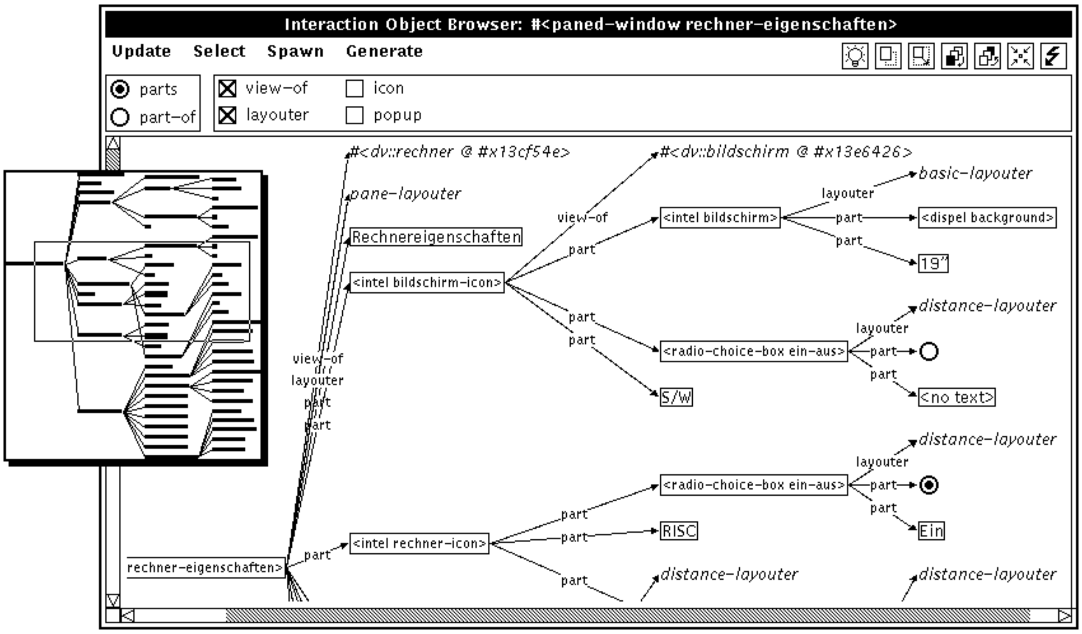

Abb. 8: Interaktionsobjekt-Browser

richtet und verändert werden. Bei Abhängigkeiten zwischen Interaktionsobjekten werden die entsprechenden Verweise durch Zeigehandlungen angegeben. So wurde etwa das in Abb. 4 dargestellte Eigenschaftsfenster mit einem in Abb. 3 konfigurierten Rechnerobjekt verknüpft, um dessen Eigenschaften darzustellen. Das Erstellen und Verknüpfen der Objekte dieses Anwendungsszenarios kann somit vollständig visuell erfolgen. Komplexere dynamische Abhängigkeiten, die nicht deklarativ über Objektattribute beschreibbar sind, werden durch Editieren von Event-Aktions-Tabellen spezifiziert und unmittelbar auf die betreffenden Objekte übertragen. Events und häufig verwendete Aktionen stehen in Pop-Up-Menüs zur Verfügung.

Der Inspektor erlaubt es, ausgehend von einem Objekt durch die Interaktionsobjekt-Hierarchie zu navigieren. Dazu kann im Property-Sheet für Struktureigenschaften (vgl. Abb. 5) durch Auswahl des hierarchisch übergeordneten Objekts (`parent`) oder eines der untergeordneten Objekte (`parts`) das zugehörige Metamenü aufgerufen werden. Diese einstufige Art der Navigation ist jedoch aufgrund des fehlenden Überblicks für komplexe Interaktionsobjekt-Hierarchien ungeeignet.

Eine graphische Visualisierung von Objektstrukturen bietet der *Interaktionsobjekt-Browser*. In Abb. 8 zeigt dieser einen Ausschnitt aus der Struktur des Eigenschaftsfensters. Im Gegensatz zu Browsern, wie sie herkömmlichen Benutzungsoberflächen-Editoren anbieten, werden nicht nur hierarchische Abhängigkeiten zwischen Interaktionsobjekten, sondern beispielsweise auch Verweise auf Layout- und Anwendungsobjekte visualisiert. Verschiedene Objekttypen und -verweise werden unterschiedlich (z. B. durch verschiedene Farben und Symbole) dargestellt. Welche Objekte und Relationen wie visualisiert werden, ist wiederum bei den Interaktionsobjekten selbst repräsentiert. Der Browser kann aus dem Inspektor aufgerufen werden; umgekehrt kann für jedes im Browser dargestellte Objekt der Inspektor aktiviert werden.

2.4 Interaktionsobjekt-Wissensbasis

Die Werkzeuge des Metasystems basieren bezüglich Präsentation und Funktionalität vorwiegend auf Wissen, das bei den Interaktionsobjekten auf Toolkit-Ebene repräsentiert ist und zur Laufzeit abgerufen werden kann. So benötigt der Editor für jedes Interaktionsobjekt Wissen darüber,

- auf welchen anderen Objekten es mittels *Drag&Drop* abgelegt werden kann und welche Aktionen dabei ausgeführt werden;
- wie ein *Prototyp* (eine Instanz mit typischen Attributwerten) oder
- eine *Kopie* (eine Instanz mit gleichen Attributwerten) erzeugt wird;
- wie *Programmcode* zur Reinstantiierung des Objekts generiert wird.

Beim Inspektor und Browser handelt es sich um Wissen,

- durch welche *Attribute* ein Objekt charakterisiert ist und wie diese visualisiert und manipuliert werden;
- welche *Operationen* auf das Objekt angewandt werden können;
- welche *Objekte* und *Relationen* wie dargestellt werden.

Teilweise ist dieses Wissen bereits implizit in den Objekten enthalten. So enthält beispielsweise eine Klassendefinition für jedes ihrer Attribute Angaben über Typ, Initialisierung und Zugriff. Abb. 9 zeigt beispielsweise die Definition einer allgemeinen Klasse für Darstellungsobjekte (`display-object`) und eine ihrer Methoden (`switch-colors`). Aus der Typinformation (`:type`) wird durch Abbildungsregeln ein geeignetes Interaktionsobjekt für die Darstellung im Inspektor generiert. Für die beiden Farbattribute (`background` und `foreground`) sind dies beispielsweise Menüs mit jeweils einem Feld zur freien Eingabe (vgl. Abb. 6).

```
(defclass display-object (interaction-object)
   ((background :type (or (member :none :parent-relative)
                          pixel)
                :accessor background
                :initarg :background
                :documentation "background color")
    (foreground :type pixel
                :accessor foreground
                :initarg :foreground
                :documentation "foreground color"))
   (:documentation "Abstract class for objects displayed
                    in a background and foreground color"))

(defmethod switch-colors ((object display-object))
   "Switch background and foreground color"
   (let ((saved-background (background object)))
      (setf (background object) (foreground object)
            (foreground object) saved-background))
   (display object))


(defmethod generate-object-code APPEND
                               ((object display-object))
   `(:background ,(color-name (background object))
     :foreground ,(color-name (foreground object))))

(defmethod inspector-attributes APPEND
                               ((object display-object)
                                (category (eql 'presentation)))
   '(background foreground)))

(defmethod inspector-operations APPEND
                               ((object display-object)
                                (category (eql 'presentation)))
   '(switch-colors)))
```

Abb. 9: Objektdefinitionen und Methoden der Wissenbasis

Anderes Wissen, wie z. B. die Zuordnung von Attributen und Operationen zu Eigenschaftskategorien oder die Behandlung von Attributen beim Kopieren oder bei der Code-Generierung, ist explizit über zusätzliche Methoden der betreffenden Objektklassen repräsentiert. Abb. 9 zeigt einige Beispiele für die Darstellungsobjekt-Klasse: Die Methode `generate-object-code` erweitert die Code-Generierung um Anweisungen zur Initialisierung der Farbattribute. Über `inspector-attributes` und `inspector-operations` werden die für die Klasse definierten Eigenschaften (Attribute und Methoden) dem Inspektor für Präsentationseigenschaften (`presentation`) zugeordnet.

Bei jeder Klasse wird nur objektspezifisches Wissen abgelegt. Beispielsweise muß ein Textobjekt nur „wissen“, daß es über ein Textattribut verfügt, nicht jedoch, daß der Text in einer Vorder- und Hintergrundfarbe dargestellt wird, da dieses Wissen in der hierarchisch übergeordneten Klasse `display-object` repräsentiert ist. Durch *multiple Vererbung*, *generische Funktionen* und *Methodenkombination* in CLOS wird das für ein konkretes Objekt relevante Wissen zugänglich gemacht. So werden etwa die einzelnen Bestandteile der Code-Generierung aus den über die Klassenhierarchie verteilten Methoden aufgesammelt und (durch `APPEND`) verkettet. Auf diese Weise steht das gesamte Wissen auch für neue, durch den Benutzer definierte Interaktionsobjekte zur Verfügung. Wird eine neue Objektklasse definiert, so ist das Metasystem automatisch auch auf Instanzen dieser Klasse anwendbar. Falls in der Klasse neue Eigenschaften (Attribute oder Methoden) definiert werden, sind nur wenige Methoden zu ergänzen, um diese auch im Metasystem verfügbar zu machen.

3 Erstellen ergonomischer Benutzungsoberflächen

Das im Interaktionsobjekt-Metasystem eingesetzte Wissen zielt im wesentlichen darauf ab, den Benutzungsschnittstellen-Entwickler bei seiner Arbeit möglichst wenig einzuschränken. Das bedeutet aber auch, daß dieser für die Qualität der erstellten Oberfläche bezüglich funktionaler, ästhetischer und ergonomischer Aspekte weitgehend selbst verantwortlich ist. Herkömmliche Editoren begegnen dieser Anforderung durch Beschränkung auf vorgefertigte Style-Guide-konforme Bausteine. Die konkrete Ausprägung dieser Bausteine und deren Zusammenspiel kann jedoch auch dort zu wenig attraktiven und unergonomischen Benutzungsoberflächen führen.

Der in XIT gewählte wissensbasierte Ansatz eröffnet zahlreiche Möglichkeiten zur Unterstützung des Entwicklers. Durch Erweitern der Wissensbasis um objektspezifische Bedingungen, die beim Bearbeiten von Interaktionsobjekten – automatisch oder auf Anfrage – überprüft werden, kann die *Funktionalität* einer Benutzungsschnittstelle – zumindest auf syntaktischer Ebene – gewährleistet werden. So könnte beispielsweise für jeden Button oder Menüeintrag durch Überprüfen der entsprechenden Attributwerte sichergestellt werden, daß er über eine Beschriftung und eine gültige Aktion verfügt. Auf dieselbe Weise lassen sich auch *ergonomische Gestaltungsrichtlinien* berücksichtigen. Dabei können sowohl objektspezifische Kriterien zur Anwendung kommen, wie z. B. geeignete Fontgrößen oder die Beschränkung der Anzahl von Menüeinträgen, als auch objektübergreifende Aspekte, wie etwa einheitliche geometrische Anordnung von Objekten, einheitliche Verwendung von Farben und Fonts, sinnvolle Farbkombinationen und konsistenter Einsatz der Maustasten.

Um den Entwickler nicht in seinen Gestaltungsmöglichkeiten einzuschränken, sollte dieses Wissen nur auf Anfrage angewandt werden. Gefundene Mängel können entweder passiv über einer Kritikerkomponente [1] aufgedeckt oder aktiv bei den betreffenden Objekten behoben werden. So verfügt der Interaktionsobjekt-Editor beispielsweise über die Möglichkeit zur automatischen Layoutgenerierung. Die durch direkte Manipulation erfolgte Grobanordnung von Interaktionsobjekten wird dabei auf Regelmäßigkeiten untersucht. Daraus werden entsprechende Layoutobjekte generiert, die Ungenauigkeiten in der Anordnung glätten, auf Veränderungen der Geometrie (z. B. der Größe des umgebenden Fensters) reagieren sowie auch neu eingefügte Objekte entsprechend anordnen.

Richtlinien zur ergonomischen Dialoggestaltung, wie sie etwa in den Normen DIN 66234/8 und ISO 9241/10 festgelegt sind, werden zum Teil bereits durch das XIT zugrundeliegende Modell unterstützt: Die *Individualisierbarkeit* einer Benutzungsschnittstelle kann über das Metasystem erfolgen, das nicht nur dem Entwickler, sondern auch dem Endbenutzer zur Verfügung steht. Die Modellierung der Interaktionsobjekte ermöglicht die *Selbstbeschreibungsfähigkeit* einer Benutzungsoberfläche auf natürliche Weise. So kann beispielsweise jedes Objekt über sein Verhalten und die von ihm repräsentierten Anwendungsfunktionalität Auskunft geben. In XIT wird dies zur Implementierung einfacher generischer Hilfekomponenten eingesetzt. Ein einfaches, aber sehr nützliches Beispiel dafür ist eine ständig sichtbare Benutzerführungszeile (*mouse documentation line* [17]), die Angaben über das Objekt macht, über dem sich der Mauszeiger momentan befindet. Durch Voreinstellung wird dargestellt, welche Benutzereingaben (Events) möglich sind und zu welchen Aktionen diese führen; die benötigte Information wird aus dem Objekt generiert. Für jedes Objekt können auch explizit Hilfetexte angegeben werden.

4 Abschlußbemerkungen und Ausblick

Es wurde gezeigt, wie durch objektorientierte Wissensrepräsentation Werkzeuge zur visuellen Programmierung graphischer Benutzungsschnittstellen generisch erstellt werden können, die bestehende Werkzeuge bezüglich Funktionalität, Flexibilität und Erweiterbarkeit übertreffen. Neben den beschrieben Komponenten zur Visualisierung statischer Strukturen (Inspektor und Browser) umfaßt XIT$_{visual}$ auch *Tracer* zur Visualisierung dynamischer Vorgänge, wie etwa das Empfangen und Verarbeiten von Events.

Der gewählte Ansatz basiert auf einem Toolkit, der es erlaubt, Interaktionsobjekte dynamisch zu erzeugen und zu manipulieren. Er konnte mit den in COMMON LISP und CLOS verfügbaren mächtigen Sprachkonstrukten einfach realisiert werden, ist jedoch nicht auf das XIT gewählte Toolkit-Modell und dessen Implementierung beschränkt. So konnten die graphischen Werkzeuge beispielsweise in wesentlichen Teilen unter SMALLTALK-80 reimplementiert werde. In beiden Fällen war es möglich, die Benutzungsschnittstellen-Entwicklungsumgebung vollständig in die bestehende Programmierumgebung zu integrieren, um so auch von allgemeinen Programmierwerkzeugen wie Objektinspektoren, Klassenbrowsern oder Methodentracern zu profitieren. Dies ist insbesondere dann wichtig, wenn die klare Trennung zwischen Benutzungsschnittstelle und interner Funktionalität nicht mehr möglich ist, wie z. B. bei direkt-manipulativen Anwendungen.

Die fehlende Trennung zwischen Entwicklungs- und Anwendungsumgebung spiegelt sich im Verzicht auf den für herkömmliche Benutzungsoberflächen-Editoren typischen Konstruktions- und Testmodus wider. Daraus ergibt sich allerdings folgendes grundlegendes Problem [15]: Wann wird ein Interaktionsobjekt bearbeitet, und wann wird es im Sinne der Anwendung bedient? Dies wurde durch Einführen von *Metaobjekten* (z. B. Paletten, Metasystem-Button und Property-Sheets) und *Meta-Interaktionen* (z. B. Drag&Drop von/auf Metaobjekte) gelöst. Diese Meta-Benutzungsschnittstelle ist in XIT selbst implementiert und deshalb auf sich selbst anwendbar.

XIT ist eine experimentelle Entwicklungsumgebung, die im wesentlichen für prototypische Anwendungen eingesetzt wird. Das Metasystem eignet sich neben der Möglichkeit zum schnellen Erstellen und Testen von Entwurfsalternativen auch als Lernumgebung, mit der bestehende Benutzungsschnittstellen untersucht und explorativ modifiziert werden können. Ergonomische Aspekte sind bisher nur ansatzweise repräsentiert. Sie sollen jedoch im Zusammenhang mit der Erstellung einer integrierten Programmier- und Lernumgebung weiter untersucht werden.

Literatur

[1] G. Fischer, A. C. Lemke, T. Mastaglio, A. I. Morch. Using Critics to Empower Users. In *Human Factors in Computing Systems, CHI '90 Conference Proceedings*, S. 337–347, 1990.

[2] J. Foley, C. Gibbs, W. C. Kim, S. Kovacevic. A Knowledge-Based User Interface Management System. In *Human Factors in Computing Systems, CHI '88 Conference Proceedings*, S. 67–72, 1988.

[3] J. Herczeg. A Design Environment for Graphical User Interfaces. In D. J. Gilmore, R. Winder, F. Détienne (Hrsg.), *User-Centred Requirements for Software Engineering Environments*, Band 123 von *NATO ASI Series F: Computer and Systems Sciences*, S. 213–223. Springer-Verlag, Berlin - Heidelberg - New York, 1994.

[4] J. Herczeg, H. Hohl, 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*, S. 181–190, 1992.

[5] J. Herczeg, H. Hohl, M. Ressel. A New Approach to Visual Programming in User Interface Design. In *Proceedings of the Fifth International Conference on Human-Computer Interaction, HCI International '93*, S. 74–79, 1993.

[6] M. Herczeg. Eine objektorientierte Architektur für wissensbasierte Benutzerschnittstellen. Dissertation, Fakultät Mathematik und Informatik der Universität Stuttgart, Dezember 1986.

[7] M. Herczeg. USIT: A Toolkit For User Interface Toolkits. In *Proceedings of Third International Conference on Human-Computer Interaction*, S. 605–612, 1989.

[8] K. Kimbrough, L. Oren. *Common Lisp User Interface Environment*. Texas Instruments Incorporated, Dallas, TX, Juli 1990.

[9] M. A. Linton, P. R. Calder, J. M. Vlissides. InterViews: A C++ Graphical Interface Toolkit. Stanford Technical Report CSL-TR-88-358, Stanford University, Juli 1988.

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

[11] B. A. Myers, R. G. McDaniel, D. S. Kosbie. Marquise: Creating Complete User Interfaces by Demonstration. In *Human Factors in Computing Systems, INTERCHI '93 Conference Proceedings*, S. 293–302, 1993.

[12] B. A. Myers, M. B. Rossen. Survey on User Interface Programming. In *Human Factors in Computing Systems, CHI '92 Conference Proceedings*, S. 195–202, 1992.

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

[14] G. Singh, C. H. Kok, T. Y. Ngan. Druid: A System for Demonstrational Rapid User Interface Development. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, S. 167–177, 1990.

[15] R. B. Smith, D. Ungar, B.-W. Chang. The Use-Mention Perspective on Programming for the Interface. In B. A. Myers (Hrsg.), *Languages for Developing User Interfaces*, Kapitel 5. Jones and Bartlett Publishers, Boston - London, 1992.

[16] SunSoft. *Open Windows Developer's Guide 3.0, User's Guide*. Sun Microsystems, Inc., Mountain View, CA, November 1991.

[17] Symbolics, Inc., Cambridge, MA. *Symbolics Reference Manual: Programming the User Interface — Concepts*, Februar 1988.

[18] P. Szekely, P. Luo, R. Neches. Beyond Interface Builders: Model-Based Interface Tools. In *Human Factors in Computing Systems, INTERCHI '93 Conference Proceedings*, S. 383–390, 1993.

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

Jürgen Herczeg
Forschungsgruppe DRUID
Universität Stuttgart, Institut für Informatik
Breitwiesenstr. 20-22, D-70565 Stuttgart
E-Mail: herczeg@informatik.uni-stuttgart.de