Diplomarbeit

Entwurf und Implementation einer Bibliothek graphischer Dialoge für die Ein- und Ausgabe von ProSet

Oliver Pütter

Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

3. November 1995

Gutachter:

Prof. Dr. E.-E. Doberkat
Dr. B. E. Sucrow

Teil I

(Schriftliche Ausarbeitung)

Zusammenfassung

Die *Prototypingsprache* ProSet besitzt lediglich Ein- und Ausgabeoperationen, welche die von Unix vorgegebenen Standardkanäle (`stdin`, `stdout` und `stderr`) nutzen. Um eine *graphische Ein- und Ausgabe* von ProSet-Werten zu ermöglichen, wird eine ProSet-Bibliothek mit graphischen Ein- und Ausgabeoperationen konstruiert. Der Aufruf dieser Operationen basiert auf dem Konzept der *internen Kontrolle.* Die *formale Spezifikation* der graphischen Dialoge erfolgt mit Hilfe von Zustandsübergangsdiagrammen, die um eine ADA-basierte Notation ergänzt sind. Des weiteren werden ProSet-Konzepte wie *Persistenz* und *Ausnahmebehandlung* integriert. Mit der Realisierung einer *interreferentiellen Ein- und Ausgabe* werden zudem software-ergonomische Aspekte berücksichtigt.

Abstract

The *prototyping language* ProSet uses the Unix standardpipes `stdin`, `stdout` and `stderr` for input and output of ProSet-values. The main purpose of this master's thesis is the construction of a ProSet-library to allow *graphical input and output.* The execution of these graphical input and output operations is controlled *internally* by the ProSet program itself. The *formal specification* of the dialogues uses state diagrams added by a simple ADA notification. ProSet-concepts like *persistence* and *exceptionhandling* are integrated as well as software-ergonomic aspects like the *reusability* of input and output data.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Diese Diplomarbeit unternimmt erste Schritte, die Prototyping-Sprache ProSet um eine graphische Benutzungsschnittstelle zu erweitern. Dabei soll in erster Linie eine Bibliothek realisiert werden, die eine graphische Ein- und Ausgabe von ProSet-Werten mit Hilfe von Dialogfenstern ermöglicht. Mit Hilfe dieser Bibliothek soll dann die graphische Darstellung von Werten im Rahmen von graphischen Benutzungsschnittstellen und der Visualisierung von ProSet-Programmen realisiert werden. Die folgende Beschreibung dieser Arbeit entspricht im wesentlichen den Phasen des Software-Konstruktionsprozesses (vgl. u.a. [Mey88]). Im Rest des Kapitels 1 werden zunächst wesentliche Grundlagen vermittelt, die zum Verständnis der Thematik und der damit verbundenen Problematik benötigt werden. Kapitel 2 und 3 enthalten die Aufgaben- und Anforderungsdefinition dieser Arbeit. Kapitel 4, 5 und 6 dokumentieren die Entwurfsphase, während Kapitel 7 die Phase der Implementierung zusammenfaßt. Kapitel 8 beschäftigt sich mit der Implementierung formal spezifizierter Dialoge. Eine Betrachtung verwandter Arbeiten sowie eine Zusammenfassung dieser Arbeit erfolgt in Kapitel 9 und 10. Anhang A enthält eine Sammlung von Abbildungen der in dieser Arbeit konstruierten Dialoge. Eine Beispiel für die Konstruktion eines graphischen Dialoges enthält Anhang B. Die zu dieser Arbeit gehörenden Quelltexte (Teil II) sind in der Lehrstuhl-Bibliothek des Lehrstuhls 10 (Software-Technologie) der Universität Dortmund vorhanden.

1.1 Graphische Benutzungsschnittstellen

Graphische Benutzungsschnittstellen gewinnen in der heutigen Zeit immer mehr an Bedeutung. Sie erlauben dem Benutzer einen vereinfachten Umgang sowohl mit *Daten* als auch mit *Applikationen* (Programmen), sofern diese über eine entsprechende graphische Anbindung verfügen. Applikationen mit graphischer Benutzungsschnittstelle bezeichnen wir als *graphische Applikationen*.

Graphische Benutzungsschnittstellen verfügen aufgrund ihres graphischen Charakters über mächtige Abstraktionsmechanismen. Sucht man nach Gründen, die für graphische Ausdrucksmöglichkeiten sprechen, so findet man diese u. a. in Berichten aus dem Forschungsgebiet der visuellen Programmiersprachen (siehe u.a. [Rae84]).

Bei einer Gegenüberstellung von graphischen und textuellen Ausdrucksmethoden werden dort die folgenden Aspekte betrachtet:

- **Informationstechnische Aspekte:**
  - **Ausdrucksvermögen:** Bilder haben im Gegensatz zu Text ein sehr hohes Ausdrucksvermögen. Hier können mehrere räumliche Dimensionen, Schattierungen und Farben zur Informationsdarstellung verwendet werden. Texte sind demgegenüber lediglich eindimensionale Ketten von Zeichen und besitzen daher ein sehr eingeschränktes Ausdrucksvermögen.
  - **Kompaktheit:** Bilder sind aufgrund ihres hohen Kodierungsgrades sehr kompakt. Man stelle sich vor, wieviele Wörter man bräuchte, um beispielsweise ein Gesicht genau zu beschreiben. Seitenlange textuelle Beschreibungen würden nicht ausreichen, um jede Einzelheit des Gesichtes zu erfassen. Ein Bild hingegen kann in der Regel einfach und sehr genau Einzelheiten eines Gesichtes beschreiben (Beispiel: Ausweis-Foto). Aber auch Text verfügt aufgrund seines hohen Abstraktionsvermögens über einen hohen Kodierungsgrad. Versucht man beispielsweise, den Begriff Kultur zu beschreiben, so reicht in der Regel eine begrenzte Anzahl von Wörtern zur Beschreibung dieses Begriffes aus. Eine Beschreibung mit Hilfe von Bildern wäre längst nicht so präzise wie die textuelle Beschreibung.
- **Kognitive Aspekte:**
  - **Sprachenunabhängigkeit:** Bilder sind unabhängig von jeder Art der natürlichen Sprache. Ein Beispiel bilden *Icons*. Das sind kleine Piktogramme, die in der Regel graphische Objekte repräsentieren, die auf ihren minimalen Inhalt reduziert wurden und dementsprechend als verkleinerte Objekte eine hohe Informationsdichte besitzen.
  - **Erfassung:** Bilder können in der Regel schneller erfaßt werden als Text. Der Zugriff auf die Information und deren Dekodierung verläuft schneller. Beispielsweise sind Verkehrsschilder aufgrund ihres graphischen Charakters schnell erfaßbar, was auch durchaus sinnvoll ist. Wegweiser zu Notausgängen bilden ein weiteres Beispiel. Sie besitzen in der Regel eine bestimmte Graphik, die schnell wahrgenommen werden kann. Texte hingegen sind sequentielle Folgen von Wörtern und Zeichen, die nicht in ihrer Gesamtheit auf Anhieb inhaltlich erfaßt werden können. Sie müssen zunächst gelesen werden. Nicht zuletzt erfolgt die Erfassung von Bildern schneller als die von Texten, weil das menschliche Auge auf die Wahrnehmung von Mustern spezialisiert ist. Alphabete sind erst seit etwa 400 bis 600 n. Chr. bekannt (Runen), während eine Verständigung auf der Basis von Bildern schon sehr viel früher (3000 v. Chr.) mit Hilfe von Bildzeichen, Hieroglyphen und Keilschriften erfolgte. Die Kommunikation auf der Basis von Texten erfolgte also erst sehr viel später.
- **Datentechnische Aspekte:**
  - **Ordnung:** Es ist möglich, einen Text alphabetisch zu sortieren. Diese Eigenschaft vereinfacht die Suche in großen Dateien bzw. Texten. Bilder hingegen lassen sich nur sehr schwer sortieren, da es an eindeutigen, sinnvollen Sortierkriterien mangelt. Es gibt jedoch Ansätze, die eine Sortierung graphischer Objekte vorsehen (siehe [LM95]).

Rein graphische oder rein textuelle Benutzungsschnittstellen sind aufgrund der oben aufgeführten Aspekte unzureichend. Für eine graphische Benutzungsschnittstelle sprechen in erster Linie kognitive Aspekte, während datentechnische Aspekte für eine textuelle Benutzungsschnittstelle sprechen. Betrachtet man informationstechnische Aspekte, so bieten sowohl textuelle als auch graphische Darstellungsmethoden Vor- und Nachteile. Demnach ist für eine graphische Benutzungsschnittstelle eine Hybridform angebracht, die möglichst viele Vorteile der textuellen Darstellung mit denen der graphischen Darstellung vereint. Diese Hybridformen sind heute bereits realisiert. Betriebssysteme, oder auch integrierte Software-Entwicklungsumgebungen verfügen über textuelle und graphische Komponenten (Beispiel: HP-SoftBench [Ger90]).

Von nun an wollen wir unter einer *graphischen Benutzungsschnittstelle* (oder auch *graphischen Benutzerschnittstelle*) eine Schnittstelle verstehen, die sowohl graphische als auch textuelle Darstellungsmethoden in sich vereint. [Sta94] definiert graphische Benutzungsschnittstellen wie folgt:

> „*Graphische Benutzerschnittstellen* oder WIMP-(Window, Icon, Menu, Pointing Device)-Schnittstellen bestehen aus einer visuellen Anzeigeeinheit sowie fenster-, piktogramm-, menue-basierten Interaktionsformen und einem Zeigehilfsmittel. Unter einer *visuellen Anzeigeeinheit* (*Visual Display Unit* (VDU) oder *Visual Display Terminal* (VDT)) verstehen wir einen Bildschirm zur Anzeige von Informationen plus einer Tastatur zur Eingabe von Kontroll- und Manipulationsfunktionen sowie Daten."

Im Mittelpunkt dieser graphischen Benutzungsschnittstellen stehen sogenannte *Fenster* (rechteckige Bildschirmausschnitte), die es dem Benutzer u.a. erlauben, Abläufe und Ausgaben mehrerer Programme gleichzeitig zu beobachten. Während er beispielsweise in einem Fenster einen Programmtext editiert, kann er diesen in einem anderen Fenster übersetzen lassen. Weitere wichtige Elemente graphischer Benutzungsschnittstellen sind neben den Fenstern die *Dialogobjekte*, die es dem Benutzer erlauben, mit graphischen Applikationen zu kommunizieren. Diese Kommunikation beschränkt sich auf *Eingaben* (Kommandos, Daten) und *Ausgaben* (Daten), da Interaktionen zwischen Benutzer und Applikation grundsätzlich als Ein- und Ausgaben interpretiert werden können.

Dialogobjekte sind oft Dingen der realen Welt nachempfunden (Metaphorik), um unerfahrenen Benutzern den Umgang mit graphischen Applikationen durch eine einfache und intuitive Bedienung zu ermöglichen. Einige Dialogobjekte verhalten sich wie *Knöpfe*, auf die der Benutzer drücken kann oder wie *Schalter*, die der Benutzer umschalten kann. *Rollbalken* erinnern an Schieberegler, mit denen man einen Wert durch „Verschieben" einstellen kann. Die Benutzung solcher Objekte durch den Benutzer wird auch unter den Begriff *direct manipulation* gefaßt. Der Benutzer hat das Gefühl, durch seine Aktionen mit den Dialogobjekten die darunterliegenden Daten direkt zu manipulieren (vgl. [GKKZ92]).

Benutzungsschnittstellen, welche die direkte Manipulation visualisierter Informationen erlauben, werden oft mit graphischen Benutzungsschnittstellen verwechselt. Dabei ist das Konzept der direkten Manipulation umfassender als jenes graphischer Schnittstellen (vgl. [Sta94]):

1. Bestimmte Objekte (z.B. ein Terminplan bei Terminplanungssystemen) werden ständig auf dem Bildschirm verfügbar gehalten.

2. Physikalische Gesten (Zeigen, Ziehen, Positionieren) sowie die Auswahl und die direkte Ausführung von Funktionen ersetzen das Eintippen von Kommandos sowie das Sich-Erinnern der Grammatik von Kommandosprachen.

3. Die Ausführung von Operationen auf den sichtbaren Objekten ist unmittelbar sichtbar und reversibel.

Abb. 1.1 zeigt einige bekannte Dialogobjekte in einem Fenster. Im einzelnen sind dies ein *Button* (Knopf als Schaltfläche, die der Benutzer mit Hilfe der Maus oder der Tastatur bedienen kann), ein *Textfeld* (Eingabefläche für Zeichen), ein *Schieberegler* und ein *Radiobutton* (spezieller Button, der zwei Zustände besitzt und den aktuellen Zustand jeweils anzeigt). Auf die Bedeutung der einzelnen Objekte gehen wir später noch genauer ein.

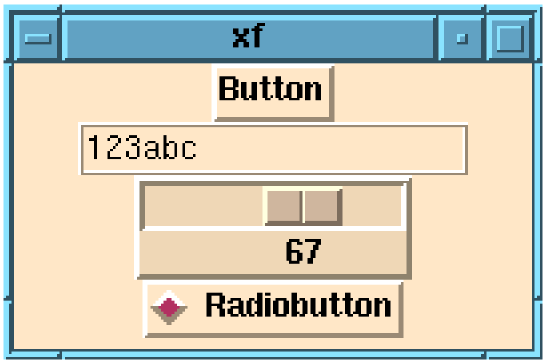


Abbildung 1.1: Dialogobjekte in einem Fenster

Das Aussehen und Verhalten der Dialogobjekte innerhalb einer graphischen Benutzungsschnittstelle sollte einheitlich bzgl. der Bedienung sein und wird als *look and feel* in *Style-Guides* (Richtlinien für graphische Benutzungsschnittstellen) festgelegt. Gottheil et al. sind folgender Meinung (siehe [GKKZ92]): „Es wäre dumm, wenn ein Schalter mal mit der linken und mal mit der rechten Maustaste bedient werden müsste.“ Verschiedene graphische Applikationen sollten ein einheitliches Aussehen und einheitliche Prinzipien für die Bedienung der Dialogobjekte besitzen (siehe u.a. [GKKZ92]).

Im folgenden Kapitel wird genauer auf die Architektur von graphischen Benutzungsschnittstellen eingegangen. Zudem wird ein Werkzeug vorgestellt, mit dessen Hilfe es dem Benutzer ermöglicht wird, graphische Applikationen einfach zu konstruieren.

1.2 X11, Motif, Tcl/Tk

X11 bezeichnet die elfte Version des *X-Window-Systems*, eines netzwerkfähigen und portierbaren Fenster-Systems, welches am Massachusetts Institute of Technology (MIT) entwickelt wurde. Es ist das Standard-Window-System für Unix und als ANSI-Standard festgeschrieben. Die Software-Schnittstelle des X-Window-Systems heißt *Xlib*, welche als Basisbibliothek des X-Window-Systems einfache Funktionen für die Entwicklung graphischer Applikationen zur Verfügung stellt. Vordefinierte Dialogobjekte, wie Menüs oder Schalter, sind nicht vorhanden. Sie müssen komplett zusammen mit ihrem Verhalten vom Benutzer implementiert werden. Aus diesem Grunde wurden aufbauend auf der Xlib vom MIT die *X-Toolkit-Intrinsics* (**Xt**-Intrinsics) entwickelt. Es handelt sich dabei um einen Mechanismus zur Erzeugung und Verwaltung von Dialogobjekten. Durch Verwendung der X-Toolkit-Intrinsics wird das Programmieren von X-Anwendungen auf eine abstraktere Ebene verlagert. Statt eines allgemeinen Fensters, in dem man alle Funktionalitäten selbst implementieren muß, hat man nun die Möglichkeit, vordefinierte Dialogobjekte mit dazugehörigen Funktionalitäten zu verwenden (siehe [BJLL91]).

Entsprechend der grundlegenden Forderung des MIT an das X-Window-System, keine Benutzungsschnittstelle festzulegen, dienen auch die Xt-Intrinsics nur als Werkzeug zum Erzeugen und Verwalten der Dialogobjekte. Welche Dialogobjekte dem Programmierer letztendlich zur Verfügung stehen, hängt vom verwendeten *Toolkit* ab (siehe [BJLL91]). Ein Toolkit besteht dabei aus einer Menge von *Widget-Beschreibungen*, die Aussehen und Verhalten der Dialogobjekte beschreiben. Je nach verwendetem Toolkit stehen dem Applikationsprogrammierer Dialogobjekte mit unterschiedlichen Funktionalitäten zur Verfügung. Mit Hilfe der Xt-Intrinsics erzeugt und verwaltet der Benutzer daraus die konkreten Dialogobjekte und stellt sie zu einer Benutzungsschnittstelle für seine Anwendung zusammen (siehe [BJLL91]). Unter anderem werden solche Toolkits zusammen mit der Fensterumgebung *Motif*, die von der Open Software Foundation (OSF) entwickelt wurde, angeboten. Insgesamt besteht *Motif* aus vier Komponenten (vgl. [GKKZ92]):

- **dem Motif-Toolkit** als Sammlung von Grundbausteinen für Applikationen mit graphischer Benutzungsschnittstelle.
  - Dialogboxen
  - Menüs
  - Schalter
  - *etc.*

  Diese Grundbausteine werden *Widgets* genannt. Sie besitzen ein dreidimensionales Erscheinungsbild und können mit der Maus oder der Tastatur bedient werden. Diese Widgets werden auch als *Motif-Widgets* bezeichnet.

- **der UIL (User Interface Language)** als Sprache, mit welcher der Benutzer die Geometrie der Applikation (u.a. Position und Größe der Fenster), ihrer Widgets und deren Aussehen beschreiben kann, ohne dabei Code in einer bestimmten Programmiersprache, wie z.B. Pascal oder C, schreiben zu müssen. Die Beschreibung wird dann zur Laufzeit der Applikation eingelesen und interpretiert. Die UIL muß nicht verwendet werden, sie ist eine Ergänzung zum Toolkit (siehe [GKKZ92]).

- **einem Motif-Window-Manager**, mit dem der Benutzer die Fenster einer Applikation interaktiv manipulieren kann. Er kann sie verschieben, vergrößern oder verkleinern. Einige Window-Manager versehen die Fenster zusätzlich mit Titelleisten und Fensterrahmen. Mittlerweile existiert eine Vielzahl von Window-Managern. Dazu gehören beispielsweise:
  - Motif-Window-Manager (mwm)
  - Tom's Virtual Tab Window-Manager (tvtwm)
  - OpenLook-Window-Manager (olwm)

  - Hewlett Packard Window-Manager

- **und dem Motif-Style-Guide** als Sammlung von Richtlinien, wie die Benutzungsschnittstelle einer Motif-Applikation aussehen und sich verhalten sollte (*look and feel*).

Abb. 1.2 zeigt die drei Ebenen unterhalb der eigentlichen Motif-Applikation (siehe [GKKZ92]). Es fällt auf, daß die Motif-Widgets die Funktionalität der darunter liegenden Schichten nicht vollständig abdecken. Einige Aufgaben, wie zum Beispiel die Ausgabe von Graphik, müssen weiterhin direkt mit Hilfe der Xlib implementiert werden, da die Xt-Intrinsics lediglich das Erzeugen und Verwalten von typischen graphischen Grundobjekten (Widgets) unterstützen (vgl. [GKKZ92]).

Der Aufbau von komplexen Benutzungsschnittstellen basiert darauf, daß man Widgets verschiedener Klassen zu komplexeren Objekten zusammenstellt. Diese zusammengesetzten Objekte sollen hier als *Formulare* bezeichnet werden.

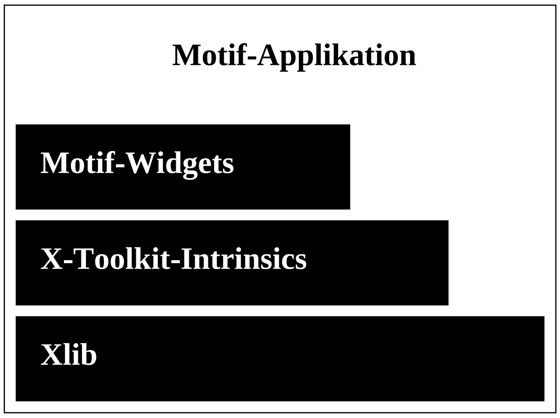


Abbildung 1.2: Die verschiedenen Ebenen einer Motif-Applikation (siehe [GKKZ92])

In sämtlichen Phasen der Software-Konstruktion greift man in der Regel auf *Werkzeuge* zurück. Das sind Programme, die den Software-Entwickler bei seiner Arbeit unterstützen und in ihrer Gesamtheit möglichst alle Phasen des Software-Konstruktionsprozesses abdecken sollten (siehe dazu u.a. [DF89]). **Tcl/Tk** ist ein Werkzeug, welches in der Phase der Anforderungsanalyse (hier im besonderen beim *Prototyping* (siehe Kapitel 1.5.1)) und der Implementierung angewandt werden kann. Es besteht im wesentlichen aus zwei Paketen: **Tcl** (**T**ool **C**ommand **L**anguage) ist eine leicht erweiterbare, zeichenkettenbasierte Skriptsprache, die ursprünglich über keinerlei graphische Anbindung verfügte. Erst durch **Tk** (**T**ool**k**it), einem Toolkit für das X-Window-System, wurde Tcl um zahlreiche Kommandos ergänzt, mit deren Hilfe es möglich ist, graphische Applikationen zu entwickeln. Der Tcl-Interpreter bedient sich einer Bibliothek von Funktionen, die in der Programmiersprache C implementiert sind. Mit Hilfe dieser Bibliotheksfunktionen kann Tcl in jede beliebige C-Applikation eingebettet und zusätzlich um applikationseigene Tcl-Kommandos erweitert werden.

Tcl/Tk zeichnet sich unter anderem durch folgende Vorzüge aus:

- **Rapid-Prototyping:** Tcl/Tk ermöglicht es, graphische Applikationen mit geringem Aufwand und in kurzer Zeit zu entwickeln, ohne auf maschinennahe Sprachen, wie z.B. C oder C++, zurückgreifen zu müssen.

- **Leichte Erlernbarkeit:** Tcl/Tk ist im Vergleich zu anderen Sprachen sehr leicht erlernbar. Dies hat verschiedene Gründe:
  - Es werden keine komplexen Datentypen (Zeiger, Record-Typen, ...) verwendet, sondern lediglich Zeichenketten (Strings).
  - Komplexe Datenstrukturen (Listen, Arrays, ...) können mit Hilfe von intuitiv verständlichen und komfortablen, aber mächtigen Operationen erzeugt und bearbeitet werden.
- **Schnelle Editier-/Debugzyklen:** Da es sich bei Tcl um eine Interpreter-Sprache handelt, entfallen lange Übersetzungsvorgänge, solange die zugrundeliegende Applikation nicht betroffen ist.
- **Interprozeßkommunikation:** Tcl verfügt über einen mächtigen *IPC-Mechanismus* (Inter-Process-Communication), der es auf einfache Art und Weise erlaubt, parallel ablaufende Applikationen zu entwickeln, die miteinander kommunizieren.

Für nähere Informationen sei auf [Ous94] verwiesen. In Kapitel 7 werden wir ebenfalls genauer auf Tcl/Tk eingehen.

1.3 Graphische Dialoge

Interaktionen zwischen Benutzer und Applikation, wie zum Beispiel die Ein- und Ausgabe von Daten, werden als *Dialoge* bezeichnet. *Graphische Dialoge* werden in der Regel mit Hilfe von *Dialogfenstern* realisiert. Das sind Fenster, die aus einem oder mehreren Dialogobjekten (Formular) bestehen und mit denen der Benutzer Ein- und Ausgaben tätigen kann. Diese Dialogfenster können aber auch zur reinen Ausgabe von Daten dienen.

Dialoge können *modal* sein. Das bedeutet, daß der Benutzer gezwungen ist, das zugehörige Formular zu bearbeiten, solange der Dialog nicht beendet ist. Andere Aktionen des Benutzers außerhalb des Dialogfensters werden ignoriert.

Modale Dialoge teilen sich im wesentlichen in zwei Untergruppen:

Applikationsmodaler Dialog: Nur Aktionen innerhalb der Applikation, die den Dialog erzeugt hat, werden ignoriert.

Systemmodaler Dialog: Sämtliche Aktionen des Benutzers außerhalb des Dialogfensters werden ignoriert.

Während eines Programmablaufes kann es durchaus zu Situationen kommen, in denen der Benutzer während eines Dialoges gehindert werden soll, andere Aktionen im System, die den Programmablauf behindern könnten, durchzuführen. In solchen Situationen wählt man in der Regel systemmodale Dialoge. Applikationsmodale Dialoge bilden eine schwächere Form der Modalität. Hier sind Aktionen außerhalb der Applikation erlaubt.

Ein Beispiel modaler Dialoge bilden Fehlermeldungen oder Warnungen, die der Benutzer erst bearbeiten (zur Kenntnis nehmen) muß, bevor er mit der Bedienung des Programmes, welches die Fehlermeldung erzeugt hat, fortfährt.

1.4 Interne vs. externe Kontrolle

Die Steuerung des Kontrollflusses einer Applikation kann im wesentlichen auf zwei Arten geschehen. Man unterscheidet die *interne Kontrolle* und die *externe Kontrolle*. Letztere hat mit der Entwicklung von graphischen Benutzungsschnittstellen enorm an Bedeutung gewonnen (siehe [GKKZ92]). Um den Unterschied beider Kontrollarten zu verstehen, sollen im folgenden zunächst diese beiden Kontrollarten erklärt werden.

Interne Kontrolle: Bei der internen Kontrolle wird der Benutzer durch das Programm geführt. Er wird zu einer bestimmten Reihenfolge der Bedienung gezwungen; jeder Schritt wird vom Programm vorgeschrieben. Programme, die auf dem Konzept der internen Kontrolle beruhen, sind normalerweise sehr starr und unflexibel, aber in der Regel leichter zu programmieren (siehe [GKKZ92]).

Externe Kontrolle: Bei der externen Kontrolle hat der Benutzer die Freiheit, die Reihenfolge seiner Aktionen selbst zu bestimmen. Er besitzt selbst die Kontrolle über das Programm und entscheidet, was als nächstes geschehen soll. Man muß sich darüber klar sein, daß diese Art der Steuerung für den Benutzer zwar komfortabel ist, für den Programmierer hingegen ist es eine „erschreckende Vorstellung“ (siehe [GKKZ92]). Dadurch, daß Aktionen vom Benutzer zu jedem Zeitpunkt ausgeführt werden können, entsteht in der Regel ein komplexer Kontrollfluß, der für den Programmierer schwer durchschaubar werden kann.

Der Unterschied zwischen interner und externer Kontrolle soll am folgenden Beispiel verdeutlicht werden.

Beispiel: Ein Programm soll folgende Gleichung lösen:

$$z = 2x + y$$

Der Benutzer legt durch seine Eingabe die Werte für x und y fest. Das Ergebnis wird schließlich berechnet und ausgegeben.

Bei einer internen Kontrolle wäre der Benutzer gezwungen, bspw. zuerst den Wert x und dann den Wert y zu belegen. Er hat nicht die Wahl, zu bestimmen, welchen Wert er zuerst eingeben möchte und wird zu einer Reihenfolge der Eingabe, und damit der Bedienung des Programmes gezwungen.

Bei einer externen Kontrolle hat der Benutzer die Wahl, welchen Wert er zuerst eingeben möchte. Das Programm reagiert entsprechend auf die Eingaben des Benutzers.

Wie wird die externe Kontrolle nun programmtechnisch realisiert, ohne daß ein komplexer, undurchschaubarer Kontrollfluß entsteht?

Die Antwort liegt in der sogenannten *Event-Schleife*. Graphische, X11-basierte Applikationen, die das Konzept der direkten Manipulation unterstützen, besitzen eine derartige Event-Schleife und basieren dementsprechend auf dem Konzept der externen Kontrolle. Hierbei werden u.a. Aktionen des Benutzers in *Ereignisse* (auch *Events* genannt) des Systems umgesetzt. Events sind Ereignisse, die rechnerintern oder vom Benutzer erzeugt werden können. Für derartige Events können *Event-Handler* definiert werden, die als Programmfragmente dann beim Eintreten der jeweiligen passenden Events aufgerufen werden. Sie bedienen den Benutzer, dessen Eingaben als Menge von Ereignissen interpretiert werden (siehe [Sta94]).

Unter anderem können folgende Events erzeugt werden:

- **ButtonPressEvent:** Ein Mausknopf wird gedrückt.
- **ExposeEvent:** Die Geometrie des Fensters wird verändert (Verschieben, Vergrößern und Verkleinern von Fenstern).
- **TimerEvent:** Ein bestimmtes Zeitintervall wurde überschritten.
- **KeyPressEvent:** Eine Taste der Tastatur wurde gedrückt.

Charakteristisch für solche extern gesteuerten Applikationen ist, daß diese von selbst nicht terminieren, da es sich bei der Event-Schleife um eine Endlos-Schleife handelt. Erst wenn die Schleife durch ausgezeichnete Aktionen (`DestroyEvent`) des Benutzers verlassen wird, terminiert das Programm und die zugehörigen Fenster der Applikation werden geschlossen. Für das Fensterverwaltungssystem X sieht diese Schleife wie folgt aus (siehe [Sta94]):

```
done = false
while (!done)
      { NextEvent (my_screen, &my_events);
        switch (my_events.type) {
               case FunctionKeyPress: Handler1(...)
               case MouseButtonClick: Handler2(...)
               case MouseMove: Handler3(...)
               case CombinedKeyPress: Handler4(...)
               case DestroyEvent: done = true }}
```

Während das Programm die Event-Schleife durchläuft, wartet es auf Events. Tritt ein Event ein, so wird der für dieses Event angegebene Event-Handler aufgerufen. Als Parameter werden dem Event-Handler *event-spezifische Informationen* übergeben. Event-spezifische Informationen sind beispielsweise die X- und Y-Koordinaten des Mauszeigers bei einem `MouseButtonClick`-Event.

In objektorientierten Fenstersystemen werden solche Event-Handler in der Regel als Methoden von Widget-Klassen vererbt und der Programmierer hat die Möglichkeit, neue Widget-Klassen mit spezialisierten Event-Handlern zu konstruieren (siehe [Smi91]).

Sämtliche extern gesteuerten Interaktionen zwischen dem Benutzer und der Applikation müssen demnach in Form von Event-Handlern an die entsprechenden Events gebunden werden. Für das oben beschriebene Beispiel bedeutet dies, daß das Programm im wesentlichen je einen Event-Handler für die Eingabe von x und y besitzt.

„Externe Kontrolle" und „ereignisgesteuerte Applikationen" sind also zwei Begriffe, die sehr eng miteinander verbunden sind. Auf die Problematik, die mit einer internen Kontrolle verbunden ist, wird in Kapitel 4 eingegangen. Für ausführlichere Informationen zu den in diesem Abschnitt angesprochenen Themen verweisen wir u.a. auf [GKKZ92].

1.5 Die Prototyping-Sprache ProSet

In den folgenden zwei Kapiteln soll näher auf die Programmiersprache ProSet und deren Anwendungsgebiet, das Software-Prototyping, eingegangen werden.

1.5.1 Prototyping

In der Phase der Anforderungsanalyse eines Software-Produktes existieren in der Regel Kommunikationslücken zwischen Auftraggeber (Kunde) und Auftragnehmer (Software-Konstrukteur). Beide Parteien besitzen eine eigene *Sprache*. Der Kunde verwendet die *natürliche Sprache*, während der Software-Konstrukteur mit *formalen Sprachen* arbeitet. Dementsprechend finden auch zwei Begriffsbildungen statt (*natürlichsprachliche Begriffsbildung* und *programmiersprachliche Begriffsbildung*) (vgl. [Sch85]).

Eine Aufgabe der Anforderungsanalyse ist es nun, die Begriffe zwischen Kunde und Software-Konstrukteur, von denen beide Parteien ihre eigenen Vorstellungen besitzen, eindeutig zu definieren. Eine Möglichkeit dazu besteht in der Konstruktion eines *Prototypen*, welcher das zu entwickelnde System repräsentiert, dessen Funktionalität jedoch auf die wichtigsten Funktionen beschränkt ist. Dieser Prototyp sollte in relativ kurzer Zeit und ohne großen Aufwand konstruiert werden können. Der Auftraggeber kann dann mit dem Prototypen experimentieren und daraus eventuelle Änderungen und Wünsche ableiten.

Es gibt verschiedene Arten des Software-Prototyping. Unter anderem können folgende Arten des Prototyping unterschieden werden:

throwaway-**Prototyping:** Der Prototyp wird nach seiner Anwendung nicht weiterverwendet. Das abzuliefernde Software-Produkt wird komplett neu entwickelt. Es werden lediglich die bei der Konstruktion des Prototypen gewonnenen Erkenntnisse genutzt. Diese Form des Prototyping kann teilweise mit den Aktivitäten einer sogenannten *Machbarkeitsstudie* verglichen werden. In einer Machbarkeitsstudie stellt man sich die Frage, ob das gewünschte Ziel (hier: das zu konstruierende Software-Produkt) überhaupt realisierbar ist. Die Entwicklung eines Prototypen kann u.a. diese Frage beantworten (vgl u.a. [DF89]).

evolutionary-**Prototyping:** Der als gut erachtete Prototyp wird vollständig zum Endprodukt ergänzt. Sämtliche während der Entwicklung dieses Prototypen erbrachten Leistungen gehen vollständig in das Endprodukt ein.

1.5.2 Eigenschaften von ProSet

ProSet (Prototyping with Sets) ist eine prozedurale Prototyping-Sprache, welche auf der Theorie endlicher Mengen basiert und folgende Datentypen zusammen mit typischen Operationen zur Verfügung stellt:

- **Der ProSet-Wert** `om`
  Dieser Wert wird immer dann verwendet, wenn unerlaubte Zugriffe erfolgen, bspw. ein Zugriff auf nicht initialisierte Variablen oder auf undefinierte Elemente eines Tupels (siehe **Zusammengesetzte Datentypen**).

- **Primitive Datentypen**
  1. `integer`: Ganze Zahlen (z.B.: `-30, 0, 540, ...`)
  2. `real`: Fließkommazahlen (z.B.: `-2.45, 0.0, 34.8754, ...`)
  3. `boolean`: `#true` und `#false`
  4. `string`: Beliebig lange Zeichenketten (z.B.: `""`, `"Hallo", ...`)
  5. `atom`: Weltweit einheitliche Werte
- **Zusammengesetzte Datentypen**
  1. `tuple`: Geordnete Folge von Objekten (z.B.: `[1.0,"abc",[1,2,3],{2,3},#true,...]`)
  2. `set`: Endliche mathematische Mengen (z.B.: `{1,"a",[1,2],{"b"},...}`)
- **Datentypen höherer Ordnung**
  1. `function`: Prozeduren mit Bürgerrechten erster Klasse
  2. `modtype`: Module mit Bürgerrechten erster Klasse
  3. `instance`: Instanziierte Module

Die Programmiersprache ProSet zeichnet sich durch ein hohes expressives Niveau aus. Sie kann in frühen Phasen der Software-Entwicklung eingesetzt werden, um die Kluft zwischen den teilweise unscharfen und vagen Vorstellungen der Kunden und der exakt zu formulierenden Anforderungsdefinition der Software-Entwickler zu überwinden. Die im Rahmen des Entwurfsprozesses häufig notwendige Rückkopplung zum Kunden wird durch Prototypen im Sinne von ausführbaren Modellen oder Spezifikationen gestaltet. ProSet, basierend auf Konzepten der endlichen Mengenlehre, erlaubt Software-Entwicklern, formale Prototypen knapp und in Anlehnung an die in der Mathematik üblichen Notation zu erstellen (siehe [FGH+93]). ProSet verfügt u.a. über folgende Eigenschaften:

Schwache Typisierung: ProSet führt keine statische Typprüfung durch. Der Benutzer ist nicht verpflichtet, Variablen zu deklarieren. Die Typbindung einer Variablen findet zur Laufzeit statt.

Ausnahmebehandlung: ProSet verfügt über Mechanismen zur Ausnahmebehandlung (siehe dazu Kapitel 4.5).

Persistenz: Daten können persistent gemacht werden. Sie existieren auch nach Beendigung des Programms, das sie erzeugt hat, und können wiederverwendet werden. Als Datenstruktur für persistente Daten dienen sogenannte *P-Files* (siehe dazu [DFG+92]).

Module und Instanzen: ProSet unterstützt das *programming in the large*. Damit ist im wesentlichen das Programmieren auf Entwurfsebene (Modularisierung des Software-Systems) gemeint, welches in der Entwurfsphase eines Software-Konstruktionsprozesses durchgeführt wird (siehe dazu [DF89]).

ProSet-Linda: ProSet unterstützt Prozeßkommunikation, die auf dem Konzept der „Tupelräume“ (virtuelle gemeinsame Datenräume) basiert.

Textuelle Ein-/Ausgabe: ProSet beinhaltet bisher Ein- und Ausgabeoperationen lediglich auf textueller Basis. Dazu werden die von UNIX vordefinierten Ein- und Ausgabekanäle (`stdin, stdout, stderr`) verwendet. Folgende Routinen werden dazu von ProSet zur Verfügung gestellt:

1. `put()`: Textuelle Ausgabe eines beliebigen ProSet-Wertes über `stdout`
2. `get()`: Textuelle Eingabe eines beliebigen ProSet-Wertes über `stdin`

Für Interessierte sei auf [DFG+92] hingewiesen.

Kapitel 2

Ziel und Aufgabe dieser Diplom-Arbeit

Ziel dieser Diplomarbeit ist der Entwurf und die Implementierung einer Bibliothek graphischer Dialoge für die Ein- und Ausgabe von ProSet. Damit soll die graphische Darstellung von Werten im Rahmen von Benutzungsschnittstellen und der Visualisierung von ProSet-Programmen realisiert werden.

Die Arbeit gliedert sich in zwei aufeinander aufbauende Teile, wie i.f. näher beschrieben ist.

- **Standard-Dialoge zur graphischen Ein- und Ausgabe von ProSet-Werten**

  Bisherige Ideen zur Ausstattung von ProSet-Programmen mit graphischer Ein- und Ausgabe beruhen auf dem Konzept der internen Kontrolle. Zunächst sind in dieser Diplomarbeit Überlegungen dazu erforderlich, wie der Aufruf von graphischen Ein- und Ausgabeoperationen aus einem ProSet-Programm heraus im Hinblick auf diese Art der Kontrolle realisiert werden kann. Diese Überlegungen sollen für spätere Erweiterungen leicht auf das Konzept der externen Kontrolle übertragbar sein. Insbesondere ist in diesem Zusammenhang zu überlegen, wie die technische Umsetzung mittels eines Benutzungsschnittstellen-Werkzeugs realisiert werden kann, ohne sich jedoch vorerst auf interne oder externe Kontrolle festzulegen. Schließlich ist durch die Implementierung von *Standard-Dialogen* die Funktionsfähigkeit dieser Realisierung zu überprüfen. Mit Standard-Dialogen sind solche Dialoge gemeint, die es erlauben, Werte verschiedenen ProSet-Typs *einfach* darzustellen, bspw. jeweils als Inhalt eines Textfeldes in einem Fenster.

- **Komplexe Dialoge zur graphischen Ein- und Ausgabe von ProSet-Werten**

  Dieser Teil der Arbeit besteht in Überlegungen zur Architektur einer Graphik-Bibliothek und der Implementierung komplexer Dialoge, für welche eine formale Spezifikation vorgegeben wird. Für jeden textuell und graphisch darstellbaren Typ in ProSet sollen hierbei mindestens je zwei unterschiedliche Dialoge zur Ein- und Ausgabe realisiert werden.

Kapitel 3

Anforderungen

Folgende Anforderungen ergeben sich aus der Aufgabenstellung dieser Diplomarbeit:

3.1 Generelle Anforderungen

1. **Interne Kontrolle:** Der Aufruf von Ein- und Ausgabeoperationen aus der zu entwickelnden Bibliothek soll intern erfolgen. Der Benutzer eines ProSet-Programmes mit graphischer Ein- und Ausgabe ist auf die durch den Programmtext vorgegebene Reihenfolge der Bedienung festgelegt. ProSet-Programme mit graphischer Ein- und Ausgabe sind <u>nicht</u> ereignisorientiert. Der Aufruf der Bibliotheksfunktionen entspricht also bis auf den graphischen Charakter dem Aufruf der textuellen Ein- und Ausgaberoutinen `put` und `get` (siehe dazu [DFG+92]).

2. **Eingaben:** Enthält ein ProSet-Programm eine Anweisung zur graphischen Eingabe eines Wertes, so soll auf dem Bildschirm ein Dialogfenster erscheinen, in dem der Benutzer seine Eingaben machen kann. Auf Knopfdruck beendet er seine Eingabe und das ProSet-Programm läuft weiter. Eingabe-Dialoge sollen *applikationsmodal* (siehe Abschnitt 1.3) sein. Das bedeutet, daß der Benutzer bei einer Eingabeaufforderung gezwungen ist, eine Eingabe zu tätigen, bevor das Programm weiterläuft.

3. **Ausgaben:** Enthält ein ProSet-Programm eine Anweisung zur graphischen Ausgabe eines Wertes, so erscheint auf dem Bildschirm ein entsprechendes Dialogfenster und das Programm läuft ungehindert weiter. Ausgabe-Dialoge sind demnach nicht modal.

4. **Breiter Anwendungsbereich:** Die zu entwickelnde Bibliothek soll nicht nur in ProSet zur Verfügung stehen, sondern auch in anderen Sprachen, die über eine C-Schnittstelle verfügen.

3.2 ProSet-spezifische Anforderungen

1. **Leichte Anpaßbarkeit:** ProSet-Programme, die über eine textuelle Ein- und Ausgabe mittels `put` und `get` verfügen, sollen leicht um eine graphische Ein- und Ausgabe erweitert werden können. Im Idealfall sollen lediglich die textuellen Ein- und Ausgaberoutinen `put` und `get` durch graphische Ein- und Ausgaberoutinen ersetzt werden.

2. **Abbruch von Eingaben:** Der Benutzer soll bei jeder Eingabeaufforderung die Möglichkeit besitzen, die Eingabe mittels der Betätigung eines `Abbruch`-Buttons abzubrechen. Der zurückgelieferte Wert entspricht dem ProSet-Wert **om**. Zusätzlich soll eine ProSet-Ausnahme erzeugt werden, die das ProSet-Programm entsprechend behandeln kann. Durch die Möglichkeit des Abbruchs einer Eingabeaufforderung durch den Benutzer werden *optionale Eingaben* ermöglicht. Bei optionalen Eingabeaufforderungen hat der Benutzer die Wahl, ob er ein Datum eingeben möchte oder nicht. Diese Entscheidungsbefugnis des Benutzers kann jedoch nicht mit der in Kapitel 1.4 beschriebenen externen Steuerung verglichen werden, da sie die Reihenfolge der Eingaben nicht berührt.

3. **Getypte Eingaben:** Zu jeder Zeit sollen nur definierte Eingaben akzeptiert werden. Das Eingabedatum muß einem ProSet-Typen eindeutig entsprechen. Beispielsweise werden Daten wie `{1,2,` oder `hallo` nicht als Eingabe akzeptiert, da man sie keinem ProSet-Typen eindeutig zuordnen kann. Hier unterscheidet sich die textuelle Eingaberoutine `get` von den graphischen Eingabe-Routinen. Undefinierte Eingaben werden von `get` standardmäßig in Zeichenketten umgewandelt (vgl. [DFG+92, Abschnitt 12]).

4. **Ausnahmebehandlung:** In folgenden Situationen soll eine ProSet-Ausnahme erzeugt werden:

   - Schließen eines Eingabefensters über die Menü-Leiste (Eintrag: `Close`).
   - Die Umgebungsvariable `DISPLAY` ist nicht gesetzt. Dabei handelt es sich um eine Shell-Variable, die Rechner, Bildschirm und Display angibt (siehe [GKKZ92]).
   - Ein Prozeß, der ein Eingabefenster kontrolliert, empfängt ein `kill`-Signal (siehe dazu [Tan90]).
   - Der Benutzer beendet eine Eingabeaufforderung durch Drücken des `Abbruch`-Buttons (siehe Punkt 2).

3.3 Graphische Anforderungen

1. **Verschiedene Darstellungen:** Der Benutzer soll die Möglichkeit besitzen, die Darstellung der ausgegebenen Daten zu variieren. So soll es beispielsweise möglich sein, eine Menge ganzer Zahlen als Tortendiagramm auszugeben und bei Verlangen auf Knopfdruck die aktuelle Darstellung in ein Balkendiagramm zu transformieren. Die Idee zu dieser Anforderung wurde durch [HH92] motiviert. Graphische Benutzungsschnittstellen, die diese Art der Transformation unterstützen, werden als *Multiple View Interfaces* bezeichnet.

3.4 Weitere Anforderungen

1. **Online-Hilfe:** Jedes Dialogfenster zur Eingabe eines ProSet-Wertes soll einen `Hilfe`-Button besitzen. Sobald der Benutzer diesen Button drückt, erscheint ein Textfenster, in dem sich ein erklärender Text zur Bedienung des Eingabe-Dialoges befindet. Dieser Hilfe-Dialog ist applikationsmodal.

2. **Speicherung von Ein- und Ausgaben:** Daten, die im Laufe eines Programmdurchlaufes ein- oder ausgegeben werden, sollen gespeichert und auf Wunsch des Benutzers neu angezeigt werden können. Dabei soll nicht nur das Datum selbst angezeigt werden, sondern auch das zugehörige Dialogfenster, mit dem das Datum ein- bzw. ausgegeben wurde.

3. **Anzahl der Dialogfenster:** Die Anzahl der gleichzeitig dargestellten Dialogfenster soll aus Gründen der Übersichtlichkeit begrenzt sein. Die Ausführung eines Programmes, welches bspw. tausend graphische Ausgaben erzeugt, sollte keine gleichzeitige Ausgabe von tausend Dialogfenstern ermöglichen.

4. **Interreferentielle Ein- und Ausgabe:** Bei jeder Eingabeaufforderung an den Benutzer, soll dieser die Möglichkeit besitzen, alte Ein- bzw. Ausgabedaten auszuwählen, statt ein neues Datum einzugeben. Die zur Verfügung gestellten Daten werden anhand des Typs der Eingabeaufforderung ausgewählt. Beispielsweise werden bei einer Eingabeaufforderung, die einen ganzzahligen Wert erwartet, auch nur ganzzahlige Werte zur Verfügung gestellt. Ein *Copy-and-Paste*-Mechanismus soll diese Anforderung realisieren. Unnötige Wiederholung von möglicherweise komplexen Eingaben desselben Datums bleiben damit dem Benutzer erspart. Diese Art der Wiederverwendbarkeit von Daten wäre ohne die Speicherung von Ein- und Ausgaben (s. Punkt 2) nicht möglich. Die interreferentielle Ein- und Ausgabe ist motiviert durch [Her94].

5. **Persistenz:** Daten, die während eines Programmablaufes ein- bzw. ausgegeben werden, sollen auch nach Beendigung eines Programmes weiterhin zur Verfügung stehen und somit wiederverwendbar gemacht werden.

3.5 Werkzeugunterstützung

Die Realisierung der Dialoge soll mittels eines Benutzungsschnittstellen-Werkzeugs geschehen. Als Werkzeug wurde **Tcl/Tk** gewählt. Hierauf werden wir in Kapitel 7 (Phase der Implementierung) näher eingehen.

Für Tcl/Tk existiert ein interaktiver *Interface-Builder* namens *Xf*. Dabei handelt es sich um ein Werkzeug, mit dem es möglich sein soll, einfach und elegant, Tcl-basierte, graphische Applikationen zu konstruieren (vgl. [Ous94]). Nachdem der Benutzer seine Schnittstelle graphisch entworfen und getestet hat, erzeugt Xf ein zugehöriges *Tcl-Skript* (siehe Kapitel 7), welches die Schnittstelle generiert.

Wir haben uns entschlossen, dieses Werkzeug nicht zu verwenden, da unserer Meinung nach Tcl/Tk keinerlei weiterer Werkzeugunterstützung bedarf. Sämtliche Tcl-Kommandos zur Erzeugung von Widgets sind ausreichend in den Manual-Pages beschrieben sowie leicht und intuitiv zu benutzen. Zudem ist die erwartete Unterstützung eines Builders zur Konstruktion graphischer Schnittstellen bei Xf nicht ausreichend gegeben.

3.6 Style-Guide

Da zu Beginn dieser Diplomarbeit keinerlei graphische Anbindungen an ProSet existieren und keine eindeutig definierten Richtlinien für graphische Schnittstellen hinsichtlich der

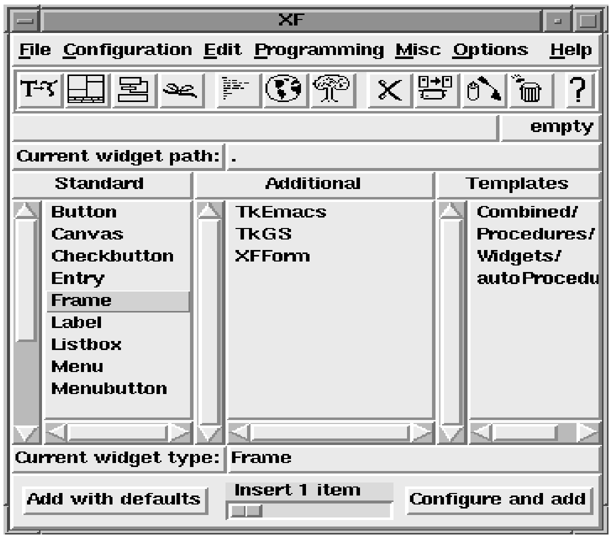


Abbildung 3.1: Der Interface-Builder XF

ProSet-Umgebung vorgegeben sind, werden im wesentlichen die Richtlinien des *Motif-Style-Guides* (siehe dazu [Fou90]) übernommen.

Kapitel 4

Konzeptioneller Entwurf

4.1 Probleme bei der Realisierung der internen Kontrolle

In Kapitel 1.4 wurde bereits erwähnt, daß die externe Kontrolle mittels Events realisiert werden kann. Applikationen, die in eine graphische und ereignisorientierte Umgebung eingebettet werden sollen, können im Hinblick auf diese Art der Kontrolle realisiert werden. Auf den ersten Blick scheint eine Realisierung der *internen* Kontrolle von graphischen Dialogen mit Hilfe von Dialogfenstern unter einer ereignisorientierten Fensterumgebung wie X11 nicht durchführbar. Um die Problematik zu veranschaulichen, werden im folgenden einige naive Ansätze zur Realisierung einer internen Kontrolle vorgestellt.

Aus Gründen der Verständlichkeit wird in den folgenden Ansätzen keine konkrete Programmiersprache benutzt, sondern vielmehr eine Pseudo-Programmiersprache, um die Konzepte, die hinter den Ansätzen stehen, hervorzuheben, ohne sie durch unnötige komplexe Programmkonstrukte zu verschleiern. Zudem beschränken wir uns der Einfachheit halber auf Dialoge zur graphischen Ausgabe. Wie später zu sehen ist, können die in diesem Kapitel vorgestellten Konzepte auch zur Realisierung graphischer Eingabe-Dialoge verwendet werden.

Ein Programm enthalte einen Aufruf `XPut()`, um eine graphische Ausgabe zu erzeugen. Dabei soll auf dem Bildschirm ein Dialogfenster mit dem auszugebenden Datum erscheinen.

Ansatz 1: Die graphische Ausgabefunktion wird als Prozedur realisiert. In dieser Prozedur wird das Motif-Toolkit *initialisiert* und das Dialogfenster erzeugt. Die Eventschleife, die durch den Aufruf der vordefinierten Funktion `XtMainLoop()` betreten wird, sorgt dafür, daß das Fenster auf dem Bildschirm erscheint und das entsprechende Datum enthält.

```
program Haupt;
...
procedure XPut(..., Datum, ...);
begin (* of XPut *)
 InitToolkit(...);
 CreateWindow(..., Datum, ...);
 XtMainLoop();
end; (* of XPut *)
...
begin (* of Haupt *)
```

```
 ...
 XPut(Ausgabe);
 ...
end. (* of Haupt *)
```

Dieser Ansatz ist aus folgenden Gründen unzureichend:

1. Da es sich bei der Event-Schleife um eine Endlosschleife handelt, werden sämtliche Anweisungen nach `XPut(Ausgabe)` nicht mehr erreicht. Das Programm kann nicht selbstständig terminieren.
2. Um das Programm terminieren zu lassen, muß der Kontrollfluß die Event-Schleife verlassen. Das hat aber zur Folge, daß sämtliche ab diesem Zeitpunkt eintretenden Events nicht mehr bearbeitet werden können. Erst beim erneuten Betreten der Event-Schleife würden eintretende Events wieder bearbeitet werden. Das so erzeugte Dialogfenster kann demnach zeitweise Zustände besitzen, in denen eine Bearbeitung des Fensters unmöglich ist. Diese Eigenschaft ist unserer Meinung nach inakzeptabel.
3. Das Programm enthält lokale Eventschleifen, also potentiell unendliche Schleifen, die im Hauptprogramm nicht sichtbar sind, was als unsauberer Programmierstil gilt (siehe [GKKZ92]).

Ansatz 2: Die graphische Ausgabeprozedur wird ebenfalls als Prozedur realisiert. Der Aufruf der Event-Schleife wird in das Hauptprogramm verlagert.

```
program Haupt;
...
procedure XPut(.., Datum, ...);
begin (* of XPut *)
 InitToolkit(...);
 CreateWindow(..., Datum, ...);
end; (* of XPut *)
...
begin (* of Haupt *)
 ...
 XPut(Ausgabe);
 XtMainLoop();
 ...
end. (* of Haupt *)
```

Dieser Ansatz ist ebenfalls unzureichend:

1. Es stellt sich folgende Frage: Wo soll der Aufruf der Event-Schleife geschehen?

   Setzt man den Aufruf an das Ende des Haupt-Programmes, werden sämtliche Anweisungen des Programmtextes bis zur Event-Schleife am Ende des Programmes ohne Verzögerung ausgeführt (Wir wollen hier von *Quasi-Terminierung* sprechen).

Allerdings erscheinen sämtliche graphische Ausgaben auch erst mit Betreten der Event-Schleife am Ende des Programmablaufes. Aufruf und Darstellung der graphischen Ausgabe sind also nicht synchron.

Wird die Event-Schleife nicht an das Ende des Programmes sondern jeweils hinter den Aufruf von `XPut()` gesetzt, ergeben sich die gleichen Probleme wie in Ansatz 1. Das Programm terminiert nicht selbständig, bevor die letzte Event-Schleife verlassen wird. Zudem würde das Hauptprogramm mehrere Event-Schleifen besitzen, was der X-Philosophie widerspricht (Jede X11-Applikation besitzt eine Event-Schleife am Ende des Programmes).

2. Der Aufruf von `XPut()` und das Erscheinen des Dialogfensters auf dem Bildschirm sind nicht synchronisiert. Der Programmierer muß selbst Aufruf (`XPut()`) und Darstellung (`XtMainLoop()`) seiner Ausgabe bestimmen. Die graphische Darstellung einer Ausgabe sollte aber unmittelbar mit dem entsprechenden Aufruf geschehen.
3. Die Erweiterung alter Programme mit textueller Ausgabe um eine graphische Ausgabe ist schwieriger. Es reicht nicht, die Aufrufe zur textuellen Ausgabe durch Aufrufe zur graphischen Ausgabe zu ersetzen. Eine oder mehrere Event-Schleifen müssen zusätzlich an geeignete Stellen des Hauptprogrammes plaziert werden, was den Programmierer dazu zwingen würde, sich mit der Fenster-Programmierung auseinanderzusetzen. Dies widerspricht allerdings den Anforderungen in Kapitel 3 (Leichte Anpaßbarkeit).

Betrachtet man die beiden Ansätze zusammen mit den Anforderungen aus Kapitel 3, so lassen sich im wesentlichen drei Folgerungen daraus ableiten:

1. Der Aufruf von graphischen Ein-/Ausgaberoutinen und die entsprechende Ausgabe der Dialogfenster auf dem Bildschirm sollen gleichzeitig erfolgen (Synchronisation von Aufruf und Darstellung).
2. Das Programm soll ohne unnötiges Eingreifen des Benutzers vollständig terminieren. Dabei soll auch eine Quasi-Terminierung ausgeschlossen werden.
3. Sämtliche Dialogfenster sollen vom Zeitpunkt ihrer Erzeugung bis hin zum Zeitpunkt ihrer Zerstörung jederzeit einen *lebendigen Zustand* besitzen, d.h. sämtliche während der Darstellungsdauer eines Dialogfensters eintretenden Events sollen bearbeitet werden können.

Die drei Punkte lassen sich nur dann realisieren, wenn es dem Programm, welches graphische Ein- und Ausgaben erzeugt, möglich ist, gleichzeitig die Events eines oder mehrerer unabhängiger Dialogfenster zu verarbeiten <u>und</u> sämtliche der graphischen Ein- bzw. Ausgabeanweisungen folgenden Anweisungen zu bearbeiten.

4.2 Realisierung der internen Kontrolle für ProSet

Im folgenden Teil wird nun beschrieben, wie sich die oben genannten Forderungen für die Realisierung der internen Kontrolle für ProSet realisieren lassen.

Eine Möglichkeit besteht in der Aufspaltung von Prozessen in mehrere unabhängige Kindprozesse. Mittels des Unix-Systemaufrufes `fork()` wird ein zum Originalprozeß exaktes Duplikat, einschließlich aller Dateideskriptoren, Register, usw., erzeugt. Nach dem Aufruf von `fork()` erfolgt der Kontrollfluß in beiden Kopien (Eltern- und Kindprozeß) getrennt voneinander. Der so erzeugte Kindprozeß führt in der Regel einen anderen Code als den des Elternprozesses aus (Beispiel: Realisierung einer Shell in [Tan90, Seite 20]).

4.2.1 Prozeßgraphen

Prozesse und deren Aufspaltung lassen sich durch sogenannte *Prozeßbäume* graphisch darstellen (vgl. [Tan90, Seite 13]). Dabei handelt es sich um Bäume, deren Wurzeln einzelne Prozesse repräsentieren. Besitzt ein Knoten A n Söhne B_1, ..., B_n, so bedeutet dies, daß der Prozeß A in n unabhängige Prozesse B_1, ..., B_n aufgespalten wurde. Wir erweitern diese Prozeßbäume zu sogenannten *Prozeßgraphen*, die sich durch folgende Erweiterungen auszeichnen:

Statements: Anweisungen, die den Zustand eines Prozesses (durch die Veränderung der Variablenbelegung) ändern, werden als *Statements* bezeichnet. Die Statements eines Programmes (Programmanweisungen), die von einem Prozeß ausgeführt werden, wollen wir durch abgerundete Kästchen, die jeweils eine Programmanweisung enthalten, visualisieren. Zwischen den Statements innerhalb eines Prozesses verlaufen wiederum Kanten, die den Kontrollfluß des zugehörigen Programmes aufzeigen.

Prozeßabspaltung: Anders als bei den oben beschriebenen Prozeßbäumen verlaufen bei den Prozeßgraphen die Kanten, die eine Prozeßabspaltung ausdrücken, von einem *Statement* des Vaterprozesses zu den erzeugten Kindprozessen. Dies bedeutet, daß ein Kindprozeß mit Ausführung des entsprechenden Statements erzeugt wird.

IPC[1]-Kanten: IPC-Kanten können als gestrichelte uni- bzw. bidirektionale Kanten zwischen zwei Knoten erscheinen. Diese Kanten beschreiben einen Nachrichtenaustausch zwischen zwei Prozessen. Empfängt ein Prozeß eine Nachricht von einem anderen Prozeß, so setzen wir voraus, daß der empfangende Prozeß vor dem sendenden Prozeß erzeugt wurde. IPC-Kanten können auch von einem Prozeßknoten zu einem Statement innerhalb eines Prozesses verlaufen. Dies bedeutet, daß der empfangende Prozeß mit Ausführung des entsprechenden Statements auf eine eintreffende Nachricht des sendenden Prozesses wartet.

Wir erhalten somit folgende Definition für Prozeßgraphen:

Ein Prozeßgraph *ProcGraph* wird dargestellt durch einen gerichteten Graphen $G = (V, E)$ mit $V = Z \cup K$, wobei

- Z die Menge der Statements einer imperativen Programmiersprache (z.B. ProSet, Tcl).
- K die Menge aller Kindprozesse.
- E die Menge aller Kanten.

[1]IPC=InterProcessComunication

Für eine Kante $(v, w) \in E$ gilt:

$$(v,w) = \begin{cases} \text{Kontrollflußkante} & \text{, falls } v, w \in Z \text{ und Statement } w \text{ folgt im} \\ & \text{Kontrollfluß des Programmes} \\ & \text{direkt nach Statement } v \\ \text{Prozeßkante} & \text{, falls } v \in Z \text{ und } w \in K \\ \text{IPC} - \text{Kante} & \text{, falls } v \in K \text{ und } w \in Z \text{ oder } v, w \in K \end{cases}$$

Die Abbildung 4.1 zeigt den Prozeßgraphen eines Programmes mit graphischer Ein- und Ausgabe.

Der Elternprozeß (ProSet-Programm mit graphischer Ein- und Ausgabe) wird bei jeder Anweisung zur graphischen Ein- und Ausgabe aufgespalten. Der jeweils erzeugte Kindprozeß besitzt die Aufgabe, das entsprechende Dialogfenster darzustellen und sämtliche bis zur Terminierung des Dialogfensters eintretenden Events zu verarbeiten, während der Elternprozeß mit der Ausführung des Hauptprogrammes fortfahren und terminieren kann. Jeder Kindprozeß repräsentiert eine typische extern gesteuerte X11-Applikation mit eigener Event-Schleife. Der Aufruf des Kindprozesses und damit der Aufruf der graphischen Ein- und Ausgaberoutinen beruht jedoch auf dem Konzept der internen Kontrolle.

Die Aufspaltung der Prozesse bei Eingabeaufforderungen erfordert einen Nachrichtenaustausch zwischen Eltern- und Kindprozeß. Die Daten, die der Benutzer mit Hilfe des vom Kindprozeß kontrollierten Dialogfensters eingegeben hat, müssen dem Elternprozeß übermittelt werden. Mit Hilfe von *IPC*-Mechanismen (Inter-Process-Communication (siehe [Tan90])) ist es möglich, diesen Nachrichtenaustausch zu realisieren. Sobald der Elternprozeß eine graphische Eingabeaufforderung ausführt, erzeugt er einen entsprechenden Kindprozeß und wartet auf eine Nachricht des Kindprozesses. Trifft eine Nachricht ein, fährt der Elternprozeß mit der Ausführung des Hauptprogrammes fort.

Der Elternprozeß terminiert mit Beendigung des Hauptprogrammes. Sämtliche vom Hauptprogramm erzeugten Kindprozesse terminieren jedoch nicht. Sie müssen vom Benutzer explizit terminiert werden.

Diese Art der Realisierung beinhaltet das Konzept der internen Steuerung und erfüllt somit Punkt 1-3 der in Kapitel 3 angegebenen generellen Anforderungen. Zudem werden sämtliche andere den Kontrollfluß betreffenden Forderungen erfüllt. Aufruf und Darstellung der graphischen Dialoge sind synchron. Da das Hauptprogramm keine eigene Event-Schleife besitzt, kommt es zu einer vollständigen Terminierung und keiner Quasi-Terminierung. Sämtliche Dialogfenster werden von eigenen Prozessen kontrolliert, so daß sie zu jedem Zeitpunkt über einen definierten Zustand verfügen.

Die folgenden zwei Kapitel beschäftigen sich näher mit der Struktur der graphischen Ein- und Ausgabeprozesse.

4.2.2 Graphische Eingabeprozesse

Wie oben bereits erwähnt, wird bei jeder Eingabeaufforderung ein entsprechender Prozeß erzeugt, der es dem Benutzer erlaubt, mit Hilfe eines erzeugten Dialogfensters seine Eingaben zu tätigen. Diese Art von Prozessen bezeichnen wir im folgenden als *graphische Eingabeprozesse*. Bei der *Zerstörung* eines Dialogfensters zur Eingabe eines ProSet-Wertes terminiert der zugehörige Prozeß und sämtliche Ressourcen (allokierter Speicher, etc.), die zur Erzeugung des Dialogfensters benötigt wurden, werden freigegeben. Da es sich bei diesen Prozessen

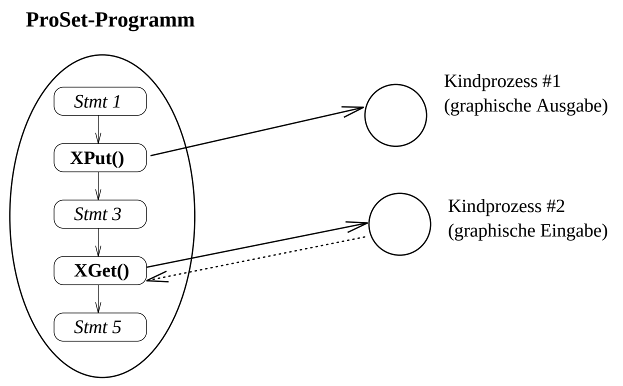


Abbildung 4.1: Prozeßgraph eines ProSet-Programmes mit graphischer Ein- und Ausgabe

um X11-basierte Applikationen handelt, besitzen sie eine Event-Schleife und somit eine Menge von Event-Handlern, die auf bestimmte Events reagieren. Das Verhalten der graphischen Eingabeprozesse soll im folgenden näher beschrieben werden:

1. Sämtliche Dialogfenster zur Eingabe eines ProSet-Wertes besitzen einen `Ok`-Button, der nur dann auswählbar ist, wenn eine *korrekte* Eingabe vorliegt. Eine Eingabe gilt genau dann als korrekt, wenn das eingegebene Datum genau einem ProSet-Typen entspricht. Drückt der Benutzer den `Ok`-Button, so wird das eingegebene Datum mittels IPC an den Elternprozeß gesendet. Gleichzeitig wird das Dialogfenster zerstört.

2. Alle Dialogfenster zur Eingabe eines ProSet-Datums besitzen einen `Abbruch`-Button. Wird dieser Button gedrückt, so wird eine entsprechende ProSet-Ausnahme erzeugt (siehe dazu Kapitel 4.5). Erst dann wird das Dialogfenster zerstört. Im Gegensatz zum `Ok`-Button ist der `Abbruch`-Button zu jeder Zeit auswählbar, d.h. die Eingabeaufforderung kann zu jeder Zeit abgebrochen werden.

3. Neben einem `Ok`- und einem `Abbruch`-Button besitzen Dialogfenster zur Eingabe eines ProSet-Wertes einen `Hilfe`-Button. Wird dieser Button gedrückt, so erscheint ein Fenster mit erklärendem Text zur Bedienung des Dialoges, welches der Benutzer wiederum mittels Knopfdruck verlassen kann. Bei dem Hilfe-Fenster handelt es sich nicht um einen Ausgabeprozeß, sondern um einen modalen extern gesteuerten Dialog des Eingabeprozesses. Der `Hilfe`-Button ist jederzeit auswählbar.

4. Schließt der Benutzer das Eingabefenster über die Menü-Leiste (Eintrag: `Close`), so wird eine entsprechende ProSet-Ausnahme erzeugt, die im zugehörigen Event-Handler spezifiziert ist.

5. Empfängt der Prozeß, der das Eingabefenster kontrolliert, ein Abbruch-Signal (`Ctrl-C`), so wird das Dialogfenster zerstört. Ein Abbruch-Signal kann sowohl vom Benutzer als auch intern vom Programm erzeugt werden.

6. Empfängt der Prozeß, der das Eingabefenster kontrolliert, ein `kill`-Signal, so wird das Dialogfenster zerstört und eine ProSet-Ausnahme erzeugt.

7. Ist die Umgebungsvariable `DISPLAY` nicht gesetzt, so wird ebenfalls eine entsprechende ProSet-Ausnahme erzeugt.

Tabelle 4.1 faßt das Verhalten von graphischen Eingabeprozessen zusammen.

Tabelle 4.1: Events mit zugehörigen Event-Handlern für graphische Eingabeprozesse

Widget/Aktionen	Event/Signal	Handleraktionen
`Ok`-Button	Button-Press-Event	Sende den eingegebenen Wert zurück und zerstöre das Dialogfenster.
`Abbruch`-Button	Button-Press-Event	Erzeuge eine ProSet-Ausnahme und zerstöre das Dialogfenster.
`Hilfe`-Button	Button-Press-Event	Erzeuge einen modalen Dialog, in dessem Fenster sich ein erklärender Text befindet.
Signalempfang	`Ctrl-C`	Zerstöre das Dialogfenster.
Signalempfang	`kill`	Zerstöre das Dialogfenster und erzeuge eine ProSet-Ausnahme.
Menüauswahl	`Close`	Zerstöre das Dialogfenster und erzeuge eine ProSet-Ausnahme.
—	`DISPLAY` nicht gesetzt	Erzeuge eine ProSet-Ausnahme.

4.2.3 Graphische Ausgabeprozesse

Wie bei der graphischen Eingabe, wird auch bei der graphischen Ausgabe von ProSet-Werten ein Prozeß gestartet. Diese Prozesse bezeichnen wir als *graphische Ausgabeprozesse*. Bei der *Zerstörung* eines Dialogfensters zur graphischen Ausgabe terminiert der zugehörige Ausgabeprozeß und sämtliche belegten Ressourcen werden freigegeben. Graphische Ausgabeprozesse besitzen folgendes Verhalten (siehe auch Tabelle 4.2):

1. Sämtliche Dialogfenster zur Ausgabe eines ProSet-Wertes verfügen über einen `Ende`-Button. Wird dieser Button gedrückt, so wird das entsprechende Dialogfenster zerstört. Der `Ende`-Button ist jederzeit auswählbar.

2. Ähnlich wie bei den Eingabeprozessen wird bei Eintreffen eines Abbruch-Signals das Dialogfenster zerstört.

3. Ist die Umgebungsvariable `DISPLAY` nicht gesetzt, so wird eine ProSet-Ausnahme erzeugt.

Tabelle 4.2: Events mit zugehörigen Event-Handlern für graphische Ausgabeprozesse

Widget/Aktionen	Event/Signal	Handleraktionen
`Ende`-Button	Button-Press-Event	Zerstöre das Dialogfenster.
Signalempfang	`Ctrl-C`	Zerstöre das Dialogfenster.
—	`DISPLAY` nicht gesetzt	Erzeuge eine ProSet-Ausnahme.

4.3 Erfüllung ergonomischer Anforderungen

Dieses Kapitel beschäftigt sich mit dem konzeptionellen Entwurf zur Erfüllung der ergonomischen Anforderungen aus Kapitel 3. Daten, die ein- bzw. ausgegeben werden, sollen gespeichert und somit wiederverwendbar gemacht werden (intereferentielle Ein- und Ausgabe). Zudem soll die Menge der gleichzeitig dargestellten Dialogfenster begrenzbar sein. Ein *Pinboard* soll diese Anforderungen erfüllen. Daten, die im Laufe eines Programmdurchlaufes ein- bzw. ausgegeben werden, werden zusammen mit allen nötigen Informationen in diesem Pinboard gespeichert und dem Benutzer jederzeit zugänglich gemacht. Gleichzeitig sollen Daten eines Programmablaufes auch nach Beendigung eines Programmes erhalten bleiben und somit wiederverwendbar sein. Das Dialogfenster des Pinboards enthält ein Menü mit folgenden Funktionen:

Anzeigen: Daten können wiederholt angezeigt werden. Dabei wird nicht nur das Datum selbst, sondern auch das zugehörige Dialogfenster, mit dem das Datum ein- bzw. ausgegeben wurde, angezeigt. Reproduzierte Dialogfenster, die alte Ein- bzw. Ausgaben enthalten, können auf Knopfdruck vom Benutzer zerstört werden.

Eingeben: Daten können mittels eines *Copy-and-Paste*-Mechanismus wiederverwendet werden. Der Benutzer kann bei einer graphischen Eingabeaufforderung ein beliebiges angezeigtes Datum des Pinboards selektieren und als Eingabe für die aktuelle Eingabeaufforderung wiederverwenden. Dabei werden nur *geeignete* Daten vom Pinboard zur Verfügung gestellt. Geeignet sind diejenigen Daten des Pinboards, die für diese Eingabe in Frage kommen. Das hängt einerseits vom Typ des einzugebenden Datums ab und andererseits von den *Darstellungsattributen* (Formular und *Dialogparameter*) des Dialoges, mit dem das Datum eingegeben werden soll. Mit Dialogparametern sind Parameter gemeint, die sich auf das Formular beziehen. Als Beispiel kann man einen Dialog betrachten, mit dessen Hilfe man einen ganzzahligen Wert eingeben kann. Die Eingabe erfolge mit Hilfe eines Schiebereglers, der eine obere und eine untere Grenze besitzt. Die Darstellungsattribute dieses Dialoges bestehen zum einen aus dem Namen des Formulars (bspw. Schieberegler-Eingabe) und zum anderen aus den Dialogparametern (obere und untere Grenze des Schiebereglers).

Speichern: Mit Hilfe dieser Funktion kann der Benutzer eine beliebige Anzahl von Daten, die während eines Programmablaufes ein- bzw. ausgegeben wurden, persistent machen. Dazu wird der Persistenzmechanismus von ProSet genutzt (siehe dazu [DFG+92, Abschnitt 9]).

Laden: Persistente Daten können mit Hilfe dieser Funktion während eines Programmablaufes zusätzlich zu den bereits vorhandenen Daten zur Verfügung gestellt werden. Sie können dann als Eingabe einer Eingabeaufforderung wiederverwendet werden. Wird diese Funktion während einer Eingabeaufforderung ausgeführt, so werden zwar sämtliche Daten des ausgewählten P-Files geladen, allerdings erscheinen zunächst nur diejenigen Daten im Pinboard, die als Eingabe des aktuellen Eingabe-Dialoges in Frage kommen.

Fensteranzahl: Mit Hilfe dieser Funktion kann der Benutzer zu jedem Zeitpunkt die maximale Anzahl der gleichzeitig dargestellten Dialogfenster zur Ausgabe eines ProSet-Wertes festlegen. Diese Beschränkung betrifft jedoch nicht die Anzahl der reproduzierten Dialogfenster. Zu Beginn eines Programmes mit graphischer Ein- und Ausgabe beträgt die Anzahl der gleichzeitig darstellbaren Ausgabefenster eins.

Entfernen: Daten, die nicht mehr benötigt werden, können aus dem Pinboard entfernt werden.

Mit Hilfe der Funktion `Alles wählen` kann der Benutzer sämtliche angezeigten Daten des Pinboards selektieren. Führt der Benutzer anschließend die Funktion `Eingeben` aus, so wird das letzte Datum der selektierten Daten ausgewählt, da nur ein Datum eingegeben werden kann. Die Funktionen `Entfernen` und `Anzeigen` berücksichtigen demgegenüber sämtliche selektierten Daten des Pinboards.

Durch das Pinboard ist es möglich, den Punkt 2 der ergonomischen Anforderungen in Kapitel 3 zu erfüllen. Eine Begrenzung der Anzahl der gleichzeitig darstellbaren Dialogfenster ist nur dann sinnvoll, wenn dadurch keine Informationen verloren gehen. Dies betrifft in erster Linie graphische Ausgaben. Würde man auf graphische Eingaben verzichten, so müßte das ablaufende Programm möglicherweise auf notwendige Daten des Benutzers verzichten und ein korrekter Ablauf des Programmes wäre in Frage gestellt. Zudem ist die Anzahl der gleichzeitig möglichen Eingaben aufgrund der Applikationsmodalität von Eingabe-Dialogen ohnehin beschränkt. Es kann maximal eine Eingabe gleichzeitig geschehen. Ein Verzicht auf Ausgaben würde demgegenüber immer noch einen korrekten Programmablauf ermöglichen, da die reine Ausgabe von Daten in der Regel für den weiteren Programmablauf aus Sicht des Programmes nicht von Interesse ist.

Das Pinboard verhindert einen Informationsverlust, da sämtliche Ein- und Ausgaben eines Programmdurchlaufes automatisch aufgezeichnet werden und auf Wunsch des Benutzers mit ihren entsprechenden Dialogfenstern neu angezeigt werden können. Aufgrunddessen ist eine Begrenzung der Ein- bzw. Ausgabeprozesse möglich und sinnvoll. Man stelle sich ein Programm vor, welches bspw. tausend Ausgaben erzeugt. Beim Ablauf dieses Programmes würden tausend Kindprozesse erzeugt. Jeder so erzeugte Prozeß erhält einen Eintrag in die *Prozeßtabelle* (Datenstruktur innerhalb eines Betriebssystems, in dem sämtliche Informationen über jeden Prozeß gespeichert werden), die aber nur über eine begrenzte Anzahl von Einträgen verfügt. Ist die Tabelle vollständig gefüllt, kann es zu Deadlock-Situationen kommen (vgl. [Tan90]).

Es stellt sich nun die folgende Frage: Wie soll das Pinboard in das bisherige Konzept der graphischen Ein- und Ausgabe integriert werden?

Die Antwort lautet wie folgt: Sobald ein Aufruf zur graphischen Ein- bzw. Ausgabe erfolgt, wird ein Kindprozeß für das Pinboard erzeugt, falls nicht zuvor bereits ein Pinboard gestartet wurde.

Bei dem Pinboard handelt es sich wie bei den graphischen Ein- und Ausgabeprozessen um eine X11-basierte Applikation.

Zwischen dem Pinboard und den Ein-/Ausgabeprozessen findet eine Interprozeß-Kommunikation wie folgt statt:

1. *Ausgabeprozesse* senden das entsprechende Datum zusammen mit Informationen über die Darstellung des Datums (Darstellungsattribute) an das Pinboard. Nachdem das Pinboard die Daten empfangen hat, überprüft es, ob dieses Datum zusammen mit seinen Darstellungsattributen bereits existiert und nimmt es gegebenenfalls in die Liste der Daten auf. Diese Liste kann also durchaus doppelte Daten enthalten; jedoch unterscheiden sich diese Daten dann zumindest in ihren Darstellungsattributen.

2. *Eingabeprozesse* senden zunächst die Darstellungsattribute des aktuellen Formulars an das Pinboard. Mit Hilfe dieser Informationen kann das Pinboard die Elemente der Datenliste filtern und, wie oben dargestellt, geeignete Daten für die Eingabe zur Verfügung stellen. Gibt der Benutzer ein neues Datum ein, so wird das eingegebene Datum zusammen mit seinen Darstellungsattributen an das Pinboard gesendet und der Filter deaktiviert, so daß der Benutzer wieder die Möglichkeit hat, sämtliche Daten des Pinboards auszuwählen. Wählt der Benutzer während des Eingabe-Dialoges ein Datum des Pinboards, so wird das gewählte Datum an den Eingabeprozeß gesendet und entsprechend in das zugehörige Fenster des aktuellen Eingabeprozesses eingefügt (im Sinne eines Copy-and-Paste-Mechanismusses). Der Benutzer hat jetzt die Möglichkeit, die gewählte Eingabe geeignet anzupassen und durch Drücken des `Ok`-Buttons zu bestätigen, was zur Folge hat, daß das eingegebene Datum zusammen mit seinen Darstellungsattributen an das Pinboard gesendet wird. Gleichzeitig wird der im Pinboard aktivierte Filter deaktiviert.

Die Kommunikation zwischen Eingabeprozeß und Pinboard kann also bidirektional sein, falls ein Kopieren aus dem Pinboard stattfindet, während die Kommunikation zwischen Ausgabeprozeß und Pinboard immer unidirektional in Richtung des Pinboards stattfindet. Abbildung 4.2 zeigt den Prozeßgraphen eines ProSet-Programmes, welches eine Anweisung zur graphischen Ausgabe und darauf folgend eine Anweisung zur graphischen Eingabe eines Datums enthält. Mit Erzeugung des graphischen Ausgabeprozesses wird ein weiterer Prozeß für das Pinboard gestartet.

Die im Pinboard gespeicherten Daten sind zeitlich geordnet. Das heißt, daß neue Daten immer an das Ende der Liste angefügt werden. Im Pinboard erscheinen sie dementsprechend immer unter dem zuletzt ein- bzw. ausgegebenen Datum. Das zuletzt eingefügte Datum wird automatisch selektiert. Um sich ein Datum zusammen mit dem entsprechenden Dialogfenster erneut anzeigen zu lassen, muß der Benutzer zunächst das entsprechende Datum selektieren und danach die Operation `Anzeigen` wählen. Dabei erscheint ein Dialogfenster, welches rein äußerlich dem ehemaligen Dialogfenster entspricht. Es handelt sich aber um ein neu erzeugtes Dialogfenster, das sich von dem Dialogfenster, welches das ausgewählte Datum an das Pinboard gesendet hat, in folgenden Punkten unterscheidet:

1. Unterschiede bei *ehemaligen* Ausgaben:

   (a) Die Titelleiste des Dialogfensters signalisiert eine alte Ausgabe.

2. Unterschiede bei *ehemaligen* Eingaben

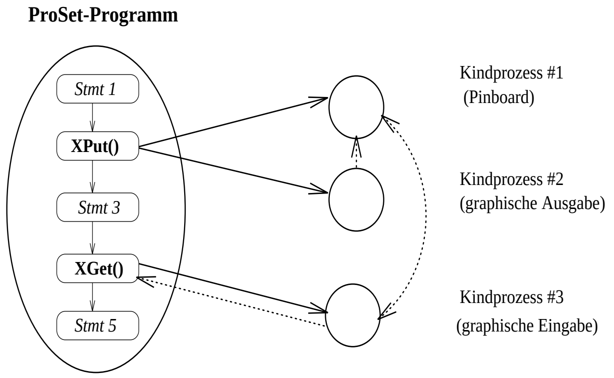


Abbildung 4.2: Prozeßgraph mit integriertem Pinboard

(a) Die Dialogfenster besitzen keinen `Ok`-Button, sondern einen `Ende`-Button.

(b) Die Titelleiste des Dialogfensters signalisiert eine alte Eingabe.

(c) Sämtliche Dialogobjekte des Dialogfensters bis auf den `Ende`-Button sind deaktiviert.

Bei den neu angezeigten Dialogfenstern handelt es sich wiederum um Kindprozesse, die jedoch auf Wunsch des Benutzers vom Pinboard gestartet werden und solange im Hintergrund laufen, bis der Benutzer sie durch ausgezeichnete Aktionen (bspw. durch Drücken des `Ende`-Buttons) zerstört. Sie sind nicht modal und es findet keine IPC-Kommunikation mit anderen Prozessen statt. Die Abbildung 4.3 zeigt einen möglichen Prozeßgraphen des Pinboards. Dabei wird jeweils durch Ausführung der Tcl-Operation `Anzeigen` ein Kindprozeß erzeugt, der das anzuzeigende Dialogfenster erzeugt und darstellt.

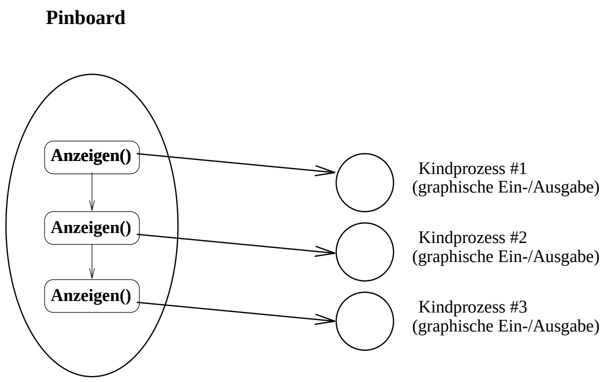


Abbildung 4.3: Prozeßgraph des Pinboards mit erneut angezeigten Dialogfenstern

4.4 Der Terminator

Sobald ein ProSet- Programm terminiert, sollten auch sämtliche Dialogfenster zerstört werden. Da es sich hier jedoch um unabhängige Unix-Prozesse handelt, bleiben die Dialogfenster auf dem Bildschirm bestehen. Zu diesen Dialogfenstern gehört auch das Pinboard. Gleichzeitig sollten bei der Terminierung eines Programmes, sämtliche Ressourcen (u.a. die Ressourcen der IPC-Verwaltung) (siehe dazu [HS87]) freigegeben werden. Nun kann es sein, daß der Benutzer nach Beendigung eines Programmes einige Ein- bzw. Ausgaben des Programmablaufes mit Hilfe des Pinboards reproduzieren möchte. Demnach dürfte das Pinboard nicht zusammen mit dem Programm terminieren, da eine Reproduktion alter Ein- und Ausgaben nur mit Hilfe des Pinboards möglich ist. Durch das ProSet-Programm generierte Ausgaben, die zum Zeitpunkt der Terminierung des Hauptprogrammes noch existieren, könnten für den Benutzer ebenfalls von Interesse sein und sollten nicht mit Beendigung des Programmes zerstört werden. Eine Lösung dieses Problems bietet der *Terminator*. Hierbei handelt es sich, ähnlich wie bei den Ausgaben, um einen Unix-Prozeß, der ebenfalls vom Hauptprogramm mit dem Aufruf des Terminators kreiert wird (Konzept der internen Steuerung).

Anders als bei graphischen Ausgaben wartet der Elternprozeß (Hauptprogramm) allerdings auf die Terminierung seines Kindprozesses (Terminator), um schließlich selbst zu terminieren.

Der Terminator erfüllt folgende Aufgaben:

1. Schließen sämtlicher aktuellen und reproduzierten Dialogfenster
2. Terminierung des Pinboards
3. Löschen der IPC-Verwaltung

Mit Hilfe des Terminators kann der Benutzer nun selbst entscheiden, zu welchem Zeitpunkt die dargestellten Dialogfenster geschlossen werden. Zudem braucht der Benutzer lediglich einen ausgezeichneten Button im Fenster des Terminators zu drücken, um sämtliche o.g. Aktionen, die vor der Terminierung des Hauptprogrammes durchgeführt werden sollten, zu starten. Dem Benutzer bleibt es somit erspart, sämtliche noch dargestellten Dialogfenster am Ende des Programmes selbst terminieren zu müssen. Der Aufruf des Terminators innerhalb eines ProSet-Programmes ist nicht zwingend. Er wird allerdings nahegelegt, da zumindest die belegten Ressourcen wieder freigegeben werden sollten. Der Aufruf kann prinzipiell an jeder Stelle des Programmtextes geschehen, sollte aber logischerweise zumindest am Ende eines Programmes mit graphischer Ein- und Ausgabe erfolgen. Wird der Terminator während des Programmablaufes gestartet, so gehen selbstverständlich mit der Terminierung des Pinboards auch sämtliche bis dahin gesammelten Daten verloren. Eine erneute Ein- oder Ausgabe hätte die Neuerzeugung des Pinboards zur Folge, das dann ohne Daten wieder angezeigt würde. Die Abbildung 4.4 beschreibt den Prozeßgraphen eines Programmes mit Terminator.

4.5 Ausnahmebehandlung

Dieses Kapitel beschäftigt sich mit den Konzepten zur Integration der Behandlung von ProSet-Ausnahmen in die Funktionen der C-Bibliothek (siehe Kapitel 4.6). Ausnahmebehandlung wird in ProSet als Mittel zur Strukturierung und Modellierung eingeführt, mit

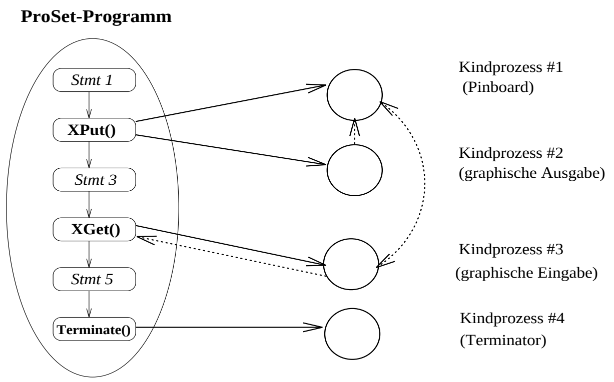


Abbildung 4.4: Prozeßgraph eines ProSet-Programmes mit Terminator

dem Ziel, einen Algorithmus möglichst knapp und präzise zu formulieren und die Ausnahmebehandlung davon zu trennen (siehe [FGH+93]). Eine *Ausnahme* beschreibt in der Regel einen undefinierten Zustand, der zur Laufzeit eines Programmes eintreten kann. Beispielsweise kann der Zugriff auf nicht existierende Daten (bspw. ein Tuple-Element mit negativem Index) eine solche Ausnahme erzeugen. Um derartige Zustände zu verarbeiten, werden sogenannte *Exception-Handler* definiert. Diese sind als ProSet-Konzept von den Event-Handlern von Motif zu unterscheiden. Bei den Exception-Handlern handelt es sich um Routinen, die bei Eintreten der entsprechenden Ausnahmen ausgeführt werden. Eine *Ausnahmebehandlung* besteht in der Regel aus folgenden Aktionen (siehe [Dob92]):

1. Tritt während der Laufzeit ein Ausnahmezustand ein, so wird eine entsprechende Ausnahme *erzeugt*. Dieses Ereignis wird nur der die Ausnahme behandelnden Einheit signalisiert.

2. Die behandelnde Einheit reagiert mit dem Aufruf der entsprechenden Handler-Routine, die explizit vom Benutzer im Programmtext angegeben werden kann. Sind keine Exception-Handler angegeben, so wird ein vordefinierter *Default-Handler* aufgerufen.

3. Die Exception-Handler haben nun die Aufgabe, die aufgetretene Situation zu bearbeiten und das System wieder in einen definierten Zustand zu bringen. Es sind zwei Formen des weiteren Programmablaufes möglich. In einem *Wiederaufnahmemodell* wird in die *auslösende Einheit* (Einheit, die die Ausnahme erzeugt hat) zurückgekehrt, so daß dort weitergearbeitet werden kann. Das *Terminierungsmodell* beendet die auslösende Einheit. Zwischen Terminierung oder Wiederaufnahme kann in ProSet dynamisch entschieden werden (siehe [FGH+93]).

Laut den Anforderungen in Kapitel 3 soll in folgenden Situationen eine ProSet-Ausnahme erzeugt werden:

1. Der Benutzer drückt den `Abbruch`-Button eines Eingabefensters.
   (Zu generierende ProSet-Ausnahme: `Cancel`)

2. Die Umgebungsvariable `DISPLAY` ist nicht gesetzt.
   (Zu generierende ProSet-Ausnahme: `NoDisplay`)

3. Der Prozeß, der ein Dialogfenster kontrolliert, empfängt ein `kill`-Signal.
(Zu generierende ProSet-Ausnahme: `Kill`)

4. Der Benutzer schließt ein Eingabefenster über die Menüleiste (Eintrag: `Close`).
(Zu generierende ProSet-Ausnahme: `Close`)

Die Mechanismen zur Ausnahmeerzeugung werden durch Aufruf von Funktionen der Laufzeit-Bibliothek von ProSet in die zu realisierende C-Bibliothek integriert. Im nächsten Kapitel werden wir genauer auf die Realisierung der Bibliotheken eingehen. Zudem sei auf das Kapitel 7 verwiesen.

4.6 Entwurf von Bibliotheken

Im Rahmen dieser Diplomarbeit werden im wesentlichen zwei Bibliotheken entwickelt, die im folgenden näher beschrieben werden.

C-Bibliothek: Diese Bibliothek enthält sämtliche C-Funktionen, die eine graphische Ein- bzw. Ausgabe von beliebigen Werten ermöglichen. Innerhalb der Funktionen werden Betriebssystemfunktionen aufgerufen, welche nötig sind, um das oben beschriebene Konzept der Prozeßaufspaltung zu realisieren. Die Mechanismen zur Behandlung von ProSet-Ausnahmen sind ebenfalls in die Funktionen der C-Bibliothek integriert.

ProSet-Bibliothek: Diese Bibliothek enthält ProSet-Funktionen, die eine graphische Ein- und Ausgabe von ProSet-Werten ermöglichen. Sie bilden im wesentlichen ProSet-spezifische Hüllen, die auf die Funktionen der C-Bibliothek zurückgreifen. Dies ist möglich, da ProSet über eine externe C-Schnittstelle verfügt. Die gekapselten Elemente der ProSet-Bibliothek verhindern, daß innerhalb der ProSet-Programme die externe C-Schnittstelle zum Aufruf der Routinen der C-Bibliothek benutzt werden muß. Durch diese Kapselung wird der Aufruf der Funktionen zur graphischen Ein- und Ausgabe wesentlich vereinfacht.

Auf eine genaue Realisierung der beiden Bibliotheken werden wir in den Kapiteln 6 und 7 eingehen.

4.7 Der Aufruf von Funktionen der C-Bibliothek

Laut den Anforderungen in Kapitel 3 soll der Aufruf von graphischen Ein- und Ausgaberoutinen sowohl aus ProSet-Programmen als auch aus Programmen heraus erfolgen können, die in einer anderen Sprache implementiert sind. Da sämtliche Bibliotheksfunktionen der C-Bibliothek in der Programmiersprache C programmiert werden, bereitet der Aufruf von graphischen Ein- und Ausgabefunktionen aus ProSet-Programmen heraus keine Schwierigkeiten, da ProSet über eine C-Schnittstelle verfügt.

Probleme bereiten jedoch die Aufrufe der Funktionen aus anderssprachigen Programmen heraus, da die Mechanismen zur Behandlung von ProSet-Ausnahmen in die C-Bibliothek integriert sind und demnach in anderssprachigen Programmen unbekannt sind. Zur Lösung dieses Problems betrachten wir zwei Vorgehensweisen:

1. Es werden zwei C-Bibliotheken realisiert, von denen eine die Mechanismen zur Erzeugung von ProSet-Ausnahmen enthält, während die andere sie nicht enthält.
2. Es wird eine C-Bibliothek realisiert, die mit Hilfe von Compiler-Optionen für den entsprechenden Anwendungsbereich übersetzt wird.

Die erste Möglichkeit ist nicht besonders elegant, da der Unterschied (Mechanismen zur Ausnahmebehandlung) zwischen beiden Bibliotheken sehr gering ist. Wir entscheiden uns demnach für die zweite Realisierung und stellen genau eine C-Bibliothek sowohl für ProSet-Programme als auch für anderssprachige Programme zur Verfügung, die eine graphische Ein- und Ausgabe von Daten ermöglicht. Abbildung 4.5 verdeutlicht folgenden Sachverhalt: ProSet-Programme greifen über die C-Schnittstelle von ProSet auf die Funktionen der C-Bibliothek zu, während C/C++-Programme einen direkten Zugriff vornehmen. Programme bzw. Bibliotheken, die in einer anderen Sprache mit externer C-Schnittstelle entwickelt wurden, können ebenfalls die Funktionen der C-Bibliothek nutzen. Der Zugriff auf die Funktionen der C-Bibliothek erfolgt hier analog zu ProSet-Programmen über die C-Schnittstelle. Zur Übersetzung der verschiedenen Bibliotheken stellen wir eine Eingabedatei für das Unix-Werkzeug `make` zur Verfügung.

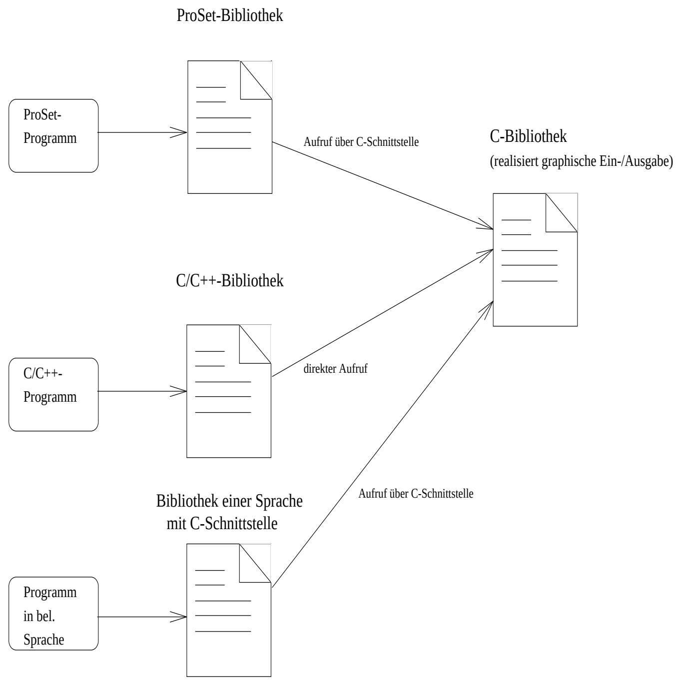


Abbildung 4.5: Aufruf von Funktionen der C-Bibliothek

Kapitel 5

Entwurf der Dialoge

In diesem Kapitel folgt nun die Beschreibung von Dialogen für die Ein- und Ausgabe von ProSet-Werten. Dabei sollte laut Kapitel 2 für einzelne Dialoge eine formale Spezifikation vorgegeben werden. Als Spezifikationssprache zur Erstellung formaler Spezifikationen war zunächst **Z** (siehe [Dil94]) geplant. Es stellte sich jedoch heraus, daß formale Spezifikationen von Dialogen mit Hilfe von **Z** schon in recht einfachen Fällen kompliziert und schwer verständlich werden und damit nicht wirklich eine Hilfe für den Entwurf von Dialogen sind.

Intuitiv verständlicher hingegen stellen sich Beschreibungen von Dialogen mit Hilfe von Zustandsübergangsdiagrammen dar. Auf eine genaue Beschreibung dieser Spezifikationsmethode soll hier jedoch nicht näher eingegangen werden. Sie folgt zusammen mit einer Bewertung der Methode in Kapitel 8.

Ideen zur Gestaltung anderer Dialoge wurden in der Regel als Prototypen realisiert, die dann inkrementell zu vollständigen Dialogen ergänzt wurden.

In den folgenden Kapiteln folgt nun eine Beschreibung der zu realisierenden Dialoge zur Ein- und Ausgabe von ProSet-Werten. Die Abbildungen der Dialogfenster enthält Anhang A.

5.1 Standard-Dialoge zur Ein- und Ausgabe von ProSet-Werten

Laut der Aufgabenbeschreibung in Kapitel 2 sollten zunächst Standard-Dialoge (einfache Dialoge) zur Ein- und Ausgabe von ProSet-Werten entwickelt werden. Dabei soll das Verhalten der Ein- und Ausgaberoutinen möglichst nicht von dem Verhalten der textuellen Ein- und Ausgaberoutinen `put` und `get` abweichen.

In Kapitel 4.2.2 wurde bereits erwähnt, daß Dialogfenster für die Eingabe von ProSet-Werten drei ausgezeichnete Buttons besitzen. Neben einem `Abbruch`- und einem `Hilfe`-Button besitzen sie einen `Ok`-Button, mit dem der Benutzer seine Eingabe beenden kann. Da laut den Anforderungen in Kapitel 3 nur getypte Eingaben akzeptiert werden sollen, liegt es nahe, den `Ok`-Button nur dann zu *aktivieren* (ein Button ist genau dann aktiviert, wenn er gedrückt werden kann), wenn eine Eingabe mit einem korrekten ProSet-Typ vorliegt. So wird der Benutzer dazu gezwungen, bei jeder Eingabeaufforderung eine korrekte Eingabe zu tätigen. Diese Entwurfsentscheidungen werden im folgenden für sämtliche Eingabe-Dialoge getroffen und deswegen nicht wiederholt beschrieben. Eine weitere generelle Entwurfsentscheidung betrifft sowohl Ein- als auch Ausgabe-Dialoge. Textfelder werden grundsätzlich mit einem horizontalen Rollbalken (siehe Kapitel 1.1) versehen, mit dem der Benutzer den sichtbaren Bereich der eingegebenen Zeichen verändern kann (für den Fall, daß die eingegebenen Zeichen die Zahl der darstellbaren Felder des Textfeldes überschreiten). Dasselbe gilt für *Listboxen* (ein Widget, welches eine Menge von Elementen darstellt und dem Benutzer erlaubt, ein oder mehrere Elemente zu selektieren), die wir standardmäßig mit einem horizontalen und einem vertikalen Rollbalken versehen wollen, um die verschiedenen Elemente der Liste sichtbar zu machen.

Dialogfenster zur Ausgabe von ProSet-Werten besitzen laut Kapitel 4.2.3 einen `Ende`-Button, mit dem der Benutzer das Fenster zerstören kann. Da der Benutzer die Möglichkeit haben sollte, Ausgabefenster zu jedem Zeitpunkt zu schließen, bleibt der `Ende`-Button jederzeit aktiv.

Aufgrund der schwachen Typisierung von ProSet liegt es nahe, Standard-Dialoge zu entwerfen, mit denen man Daten beliebigen ProSet-Typs eingeben kann. Diese sollten zudem möglichst einfach sein. Das führt zu folgenden Standard-Dialogen:

Standard-Eingabe Das Dialogfenster dieses Dialoges besitzt lediglich ein Textfeld, in welches der Benutzer seine Eingabe eintragen kann. Handelt es sich bei der Eingabe um einen Wert mit einem korrekten ProSet-Typ, so wird der `Ok`-Button aktiviert. Durch Drücken dieses Buttons wird der eingegebene Wert an das Hauptprogramm übergeben und das Dialogfenster zerstört.

Standard-Ausgabe Dieser Dialog entspricht äußerlich dem Standard-Dialog zur Eingabe eines ProSet-Wertes. Das zugehörige Dialogfenster enthält ebenfalls ein Textfeld, in welchem sich das ausgegebene Datum befindet. Durch Drücken des `Ende`-Buttons wird das Dialogfenster zerstört.

Die Ein- bzw. Ausgabe von ProSet-Werten mit Hilfe der Standard-Dialoge entspricht im weitesten Sinne der textuellen Ein- und Ausgabe mit Hilfe von `put` und `get`. Des weiteren entwerfen wir einen Eingabe-Dialog, dessen Formular dem Standard-Dialog zur Eingabe eines ProSet-Wertes entspricht aber zusätzlich mit einem benutzerdefinierten, regulären Ausdruck (siehe Kapitel 7.9) versehen wird, um die Anzahl der möglichen Eingaben einzuschränken.

5.2 Komplexe Dialoge zur Eingabe von ProSet-Werten

Dieses Kapitel beschäftigt sich ausschließlich mit graphischen Dialogen zur *Eingabe* von ProSet-Werten. Dabei werden zunächst die primitiven und schließlich die komplexen Datentypen von ProSet betrachtet.

5.2.1 Eingabe von ProSet-Werten primitiven Typs

ProSet stellt folgende primitiven Datentypen zur Verfügung:

- `Integer`
- `Real`

- `String`
- `Boolean`
- `Atom`

Auf eine graphische Ein- und Ausgabe von Werten des Typs `Atom` wurde verzichtet, da diese nicht sinnvoll erscheint. Atome werden üblicherweise intern vom Rechner selbst mit Werten belegt, um ihre Eindeutigkeit sicherzustellen.

Bei Dialogen zur graphischen Eingabe von Werten des Typs `String` mangelte es an Ideen, so daß anstelle eines Dialoges zur Eingabe von Zeichenketten ein sinnvollerer Dialog entworfen wurde.

Der Datentyp `Integer`

Im folgenden werden zwei Dialoge zur graphischen Eingabe von Werten des Typs `Integer` beschrieben.

Integer-Zehner-Block-Eingabe: Dieser Dialog soll eine Zehner-Block-Tastatur realisieren. Dabei handelt es sich im wesentlichen um eine rechteckig angeordnete Fläche von zehn Ziffern-Tasten, wie sie etwa bei einem Taschenrechner bekannt sind. Mit Hilfe dieser Tasten kann der Benutzer nun mit Hilfe der Maus eine ganze Zahl ziffernweise eingeben. Die eingegebenen Zahlen erscheinen in der Reihenfolge der Eingabe in einem Textfeld. Der Benutzer kann seine Eingabe auch mit Hilfe der Tastatur tätigen, indem er mit Hilfe der Maus das Textfeld aktiviert (bei Aktivierung eines Textfeldes erscheint in der Regel ein blinkender Cursor) und die entsprechenden Tasten auf der Tastatur betätigt.

Neben den Zifferntasten gibt es drei weitere Tasten mit folgenden Bezeichnungen:

`+/-` **:** Mit Hilfe dieser Taste kann der Benutzer das Vorzeichen der eingegebenen Zahl ändern. Sie kann wiederholt betätigt werden, was zur Folge hat, daß das aktuelle Vorzeichen jeweils invertiert wird.

`Clr` **:** Mit Hilfe dieser Taste kann der Benutzer das komplette Textfeld löschen. Ist das Textfeld bereits leer, hat das Betätigen dieser Taste keine Wirkung.

`Del` **:** Mit Hilfe dieser Taste löscht der Benutzer die zuletzt eingegebene Ziffer. Sie kann wie die Vorzeichen-Taste wiederholt betätigt werden.

Schieberegler-Eingabe: Mit Hilfe dieses Dialoges soll der Benutzer durch Verschieben eines Schiebereglers eine ganze Zahl festlegen, die er eingeben möchte. Der Schieberegler besitzt eine obere und eine untere Grenze, die der Benutzer als Parameter der zugehörigen Eingabefunktion übergeben muß. Diese werden an den entsprechenden Stellen im Dialogfenster angezeigt. Der Schieberegler läßt sich sowohl mit der Maus als auch mit der Tastatur bedienen, was je nach Differenz der Grenzwerte und damit der Schrittweite von Vorteil ist. Die Bedienung eines Schiebereglers mit der Tastatur ist im Motif-Style-Guide (siehe dazu [Fou90]) beschrieben und soll hier nicht näher erläutert werden.

Der Datentyp `Real`

Folgende Dialoge dienen zur Eingabe eines Wertes vom Typ `Real`:

Real-Zehner-Block-Eingabe Ähnlich wie im vorherigen Abschnitt soll für die Eingabe eines `Real`-Wertes eine Zehner-Block-Tastatur realisiert werden. Die zu realisierende Funktionalität entspricht im wesentlichen der oben Beschriebenen. Allerdings wird eine zusätzliche Taste realisiert, die folgende Bezeichnung trägt:

. : Mit Hilfe dieser Taste kann der Benutzer einen Dezimalpunkt eingeben. Ein Dezimalpunkt wird nur dann auf Knopfdruck in das Textfeld eingefügt, wenn sich noch keiner im Textfeld befindet.

Die Anzahl der Vor- und Nachkommastellen werden vom Benutzer im Programmtext als Parameter der zugehörigen Eingabefunktion angegeben. Der Benutzer muß sich an diese Vorgaben halten; eine Eingabe, die diese Bedingungen nicht erfüllt, kann nicht eingegeben werden (De-Aktivierung des `Ok`-Button).

Bruch-Eingabe Ein Wert vom Typ `Real` kann als gemischter Bruch eingegeben werden. Dieser Dialog realisiert eine Eingabe auf der Basis der gemischten Schreibweise einer rationalen Zahl. Dazu werden genau drei Textfelder benötigt:

- **Ganzzahliger Wert:** Als ganzzahliger Wert wird eine beliebige ganze Zahl erwartet. Wird kein Wert in das zugehörige Textfeld eingetragen, so wird 0 angenommen.
- **Zähler:** Dieses Textfeld akzeptiert jede natürliche Zahl einschließlich 0.
- **Nenner:** Dieses Textfeld akzeptiert jede natürliche Zahl, die größer als 0 ist.

Eine Eingabe für Nenner und Zähler ist obligatorisch, während eine Eingabe für den ganzzahligen Wert optional ist.

Der Datentyp `String`

Wie oben bereits angedeutet, mangelte es an Ideen für komplexe Dialoge zur Eingabe von Zeichenketten. Alternativ wurde neben einem Dialog zur Eingabe eines Dateinamens ein Dialog zur Eingabe eines beliebigen ProSet-Datums realisiert.

Wahl-Eingabe Das Dialogfenster dieses Dialoges besitzt für jeden ProSet-Typen einen Radiobutton (siehe Kapitel 1.1). Bevor der Benutzer ein Datum eingibt, muß er den Typ des Datums mit Hilfe dieser Radiobuttons festlegen. Je nach Wahl wird das zugehörige Textfeld mit entsprechenden Zeichen eingerahmt. Möchte der Benutzer beispielsweise eine Menge eingeben, so muß er zunächst den Typ `Menge` auswählen. Daraufhin wird das Textfeld mit Mengenklammern ({ }) eingerahmt und der Benutzer kann die von ihm gewünschte Menge eingeben. Während der Eingabe eines Datums kann der Benutzer mit Hilfe der Radiobuttons den Typ des Eingabedatums ändern. Entspricht das bis dahin eingegebene Datum nicht dem neu eingegebenen Typ, so bleibt der `Ok`-Button des Eingabe-Dialoges solange deaktiviert, bis der Benutzer das eingegebene Datum entsprechend angepaßt hat.

Filename-Eingabe Mit Hilfe dieses Dialoges, der im wesentlichen einer *Fileselection-Box* entspricht, kann der Benutzer einen Dateinamen eingeben. Das zugehörige Dialogfenster enthält zwei Listboxen, von denen das eine die Unterverzeichnisse des aktuellen Verzeichnisses und das andere die verfügbaren Dateien in diesem Verzeichnis auflistet. Beide Listboxen verfügen analog zu den Textfeldern über einen vertikalen und horizontalen Rollbalken, mit dem der Benutzer die Menge der sichtbaren Einträge bestimmen kann. Das aktuelle Verzeichnis wird zu jedem Zeitpunkt in einem Textfeld ausgegeben. Mit Hilfe der Maus kann der Benutzer nun durch einen Doppelklick auf ein Element der Verzeichnis-Liste in das entsprechende Unterverzeichnis wechseln. Die Menge der verfügbaren Dateien wird mit einem Verzeichniswechsel automatisch aktualisiert. Selektiert der Benutzer mit Hilfe der Maus einen Dateinamen, so kann er schließlich durch Drücken des `Ok`-Buttons seine Eingabe bestätigen. Das Eingabedatum besteht aus dem absoluten Pfad der ausgewählten Datei. Ein weiteres Textfeld gibt dem Benutzer die Möglichkeit, die Liste der angezeigten Dateien zu filtern. Dazu muß er in dem Textfeld einen entsprechenden Filter eingeben, worauf die Liste der angezeigten Dateien sofort aktualisiert wird. Bei dem angegebenen Filter handelt es sich um vereinfachte *reguläre Ausdrücke* (siehe Kapitel 7.9), deren Zeichen folgende Semantik besitzen:

*** :** Sequenz von null oder mehr beliebigen Zeichen.

? : Ein beliebiges Zeichen.

`[chars]` **:** Ein Zeichen der Menge `chars`.

Beispiele: `[1,2,3]` **:** Entweder `1`, `2` oder `3`.

`[a-z]` **:** Genau ein Zeichen aus der Menge `{‘a‘,...,‘z‘}`.

\\`x` **:** Das Zeichen `x`.

Sämtliche anderen Zeichen des regulären Ausdrucks müssen an entsprechender Stelle im Datum auftauchen, damit das Datum den regulären Ausdruck erfüllt. Weitere Informationen zu regulären Ausdrücken findet man unter anderem in [Ous94].

Beispiele:

***.c** Dieser Filter filtert sämtliche Dateien mit der Endung `.c`

***hallo.**`[ch]` Dieser Filter filtert sämtliche Dateien mit der Endung `.c` oder `.h`, deren Name mit `hallo` endet.

Der Datentyp `Boolean`

Zur Eingabe eines Booleschen Wertes wurden folgende Dialoge realisiert:

Antwort-Eingabe: Dieser Dialog dient zur Beantwortung einer Frage, die der Benutzer als Parameter der zugehörigen Eingabefunktion angibt. Das zugehörige Dialogfenster besitzt zwei Buttons, die zur Beantwortung der gestellten Frage mit *Ja* oder *Nein* dienen. Drückt der Benutzer den `Ja`-Button, so wird der Wert `#true` an das PROSET-Programm übergeben. Verneint der Benutzer durch Drücken des `Nein`-Buttons die gestellte Frage, so wird der Wert `#false` an das PROSET-Programm übergeben.

True-oder-False-Eingabe Das Dialogfenster dieses Dialoges enthält genau zwei Radiobuttons, mit dem der Benutzer entweder den Wert `#true` oder `#false` eingeben kann.

5.2.2 Eingabe von ProSet-Werten komplexen Typs

Dieses Kapitel beschäftigt sich mit den Dialogen zur Eingabe von ProSet-Werten komplexen Typs. Für die Eingabe von Mengen und Tupeln wurden folgende Dialoge entworfen:

Der Datentyp `Set`

Folgende Dialoge dienen zur Eingabe eines Datums des Typs `Set`

Regular-Set-Eingabe Mit Hilfe dieses Dialoges sollen Mengen elementweise eingegeben werden. Zur Eingabe dient wie üblich ein Textfeld und ein Button mit der Bezeichnung `Hinzufügen`. Nachdem der Benutzer ein Element der Menge mit Hilfe der Tastatur eingetragen hat, kann er dieses Element durch Drücken des `Hinzufügen`-Buttons zur Liste der bis zu diesem Zeitpunkt eingegebenen Elemente addieren. Sämtliche Elemente, die der Benutzer eingibt, erscheinen in einer Listbox. Klickt der Benutzer mit der Maus ein Element der Listbox und somit ein Element der Menge an, so wird das Element zunächst aus der Menge der Elemente entfernt und in das Textfeld eingetragen. Der Benutzer kann eventuelle Änderungen an dem Element vornehmen und gegebenenfalls das geänderte Element wieder einfügen. Klickt der Benutzer ein Element der Listbox an, so ersetzt dieses den Inhalt des Textfeldes.

Dieser Dialog ist zusätzlich mit einem *regulären Ausdruck* (s. Kapitel 7.9) behaftet, den der Benutzer als Parameter an die entsprechende Eingabe-Funktion übergibt. Nur Elemente, die durch diesen regulären Ausdruck beschrieben werden, werden als Elemente der Menge akzeptiert.

Map-Eingabe Ein Menge, die ausschließlich aus zweielementigen Tupeln besteht, wird auch als `map` (Abbildung) bezeichnet. Daten diesen Typs können als Abbildung von Elementen eines Vorbereiches in Elemente eines Nachbereiches interpretiert werden. Der Vorbereich dieser Abbildung besteht aus der Menge der ersten Elemente der Tupel und der Nachbereich aus der Menge der zweiten Elemente der Tupel. Der hier beschriebene Dialog dient zur Eingabe einer solchen Menge. Dafür übergibt der Benutzer als Parameter zwei Mengen, die Vor- und Nachbereich einer Abbildung repräsentieren. Die Elemente des Vorbereiches werden graphisch auf der linken Seite einer *Canvas* (Zeichenfläche) dargestellt, während die Elemente des Nachbereiches auf der rechten Seite dargestellt werden. Mit Hilfe der Maus muß der Benutzer zunächst ein Element des Vorbereiches anklicken und kann es dann durch Anklicken mit einem Element des Nachbereiches assoziieren. Die jeweilige Zuordnung zweier Elemente wird durch einen Pfeil dargestellt. Dieser Pfeil verbindet jeweils ein Element des Vorbereiches mit einem Element des Nachbereiches. Klickt der Benutzer mit Hilfe der Maus einen dieser Pfeile an, so wird dieser gelöscht und das zugehörige Element aus der Liste der bis dahin eingegebenen Elemente der Abbildung entfernt.

Der Datentyp `Tuple`

Im folgenden werden Dialoge zur Eingabe eines Tupels beschrieben:

Equalizer-Eingabe Mit Hilfe dieses Dialoges kann der Benutzer ein Tupel bestehend aus endlich vielen ganzen Zahlen eingeben. Dazu stehen dem Benutzer eine Menge von

Schiebereglern zur Verfügung, deren Anzahl sowie deren Grenzen als Parameter der entsprechenden Eingabe-Funktion übergeben werden. Dabei entspricht ein Schieberegler genau einem Element des Tupels. Mit Hilfe der Maus oder der Tastatur kann der Benutzer durch Verschieben der einzelnen Regler für jedes Element einen Wert einstellen. Die Parameter für die obere und untere Grenze gelten für alle Schieberegler.

Grafik-Eingabe Dieser Dialog entspricht zum Teil dem zuvor beschriebenen Dialog zur Eingabe einer Menge (`Regular-Set-Eingabe`). Er dient jedoch sowohl zur Eingabe einer Menge als auch zur Eingabe eines Tupels. Zusätzlich verfügt das zu diesem Dialog gehörende Dialogfenster über eine *Canvas* (Zeichenfläche), in der jeweils das aktuelle Datum (Tupel oder Menge), welches bis dahin elementweise eingegeben wurde, graphisch dargestellt wird. Die graphische Darstellung entspricht dabei der eines Baumes, wobei die Elemente eines Tupels bzw. einer Menge jeweils als Söhne einer Wurzel (als Symbol eines Tupels bzw. einer Menge) dargestellt werden. Mit jedem Element, welches der Benutzer entfernt bzw. hinzufügt, wird die graphische Darstellung aktualisiert. Zwei Radiobuttons mit den Bezeichnungen `Tupel` und `Menge` geben dem Benutzer die Möglichkeit, den Typ des einzugebenden Datums jederzeit festzulegen. Auch hier wird bei jeder neuen Festlegung die graphische Darstellung aktualisiert.

Die Dialoge dieses Typs sind nicht mit einem regulären Ausdruck behaftet. Der Benutzer kann demnach Mengen und Tupel beliebiger Art eingeben.

5.3 Komplexe Dialoge zur Ausgabe von ProSet-Werten

In diesem Kapitel werden Dialoge zur *Ausgabe* von ProSet-Werten beschrieben. Dabei werden analog zu Kapitel 5.2 zunächst die primitiven und schließlich die komplexen ProSet-Typen betrachtet. An dieser Stelle sei noch einmal in Erinnerung gerufen, daß laut den Anforderungen in Kapitel 3 Ausgabe-Dialoge verschiedene Darstellungen des ausgegebenen Datums unterstützen sollen (*Multiple-View-Interfaces*). Diese Anforderung betrifft im wesentlichen die Ausgabe-Dialoge für Mengen und Tupel, da die primitiven Datentypen unseres Erachtens zu wenig Informationen bieten, als daß eine änderbare Darstellung hier Sinn machen würde.

5.3.1 Ausgabe von ProSet-Werten primitiven Typs

Der Datentyp `Integer`

Folgende Dialoge sollen zur Ausgabe eines Datums des Typs `Integer` dienen:

Wahl-Ausgabe Mit Hilfe dieses Dialoges kann der Benutzer eine positive, ganze Zahl ausgeben und deren Darstellung mit Hilfe der folgenden drei Buttons verändern.

Dez: Die ausgegebene Zahl wird in eine Dezimal-Zahl umgewandelt.

Hex: Die ausgegebene Zahl wird in eine Hexadezimal-Zahl umgewandelt.

Bin: Die ausgegebene Zahl wird in eine Binär-Zahl umgewandelt.

Das Datum erscheint jeweils in dem dafür vorgesehenen Textfeld. Die Darstellung des Datums bei Erzeugung der Ausgabe entspricht der Dezimaldarstellung (Default-Darstellung).

Abakus-Ausgabe Dieser Dialog dient zur Ausgabe eines Datums des Typs `Integer`. Dabei wird die auszugebende Zahl in eine graphische Darstellung transformiert, die Ähnlichkeit mit einem Abakus besitzt. Eine ganze Zahl entspricht dabei einer Menge von farbigen Kugeln, von denen jede einen ausgezeichneten Wert besitzt. Addiert man die Werte der dargestellten Kugeln, so erhält man die ausgegebene Zahl. Die Zahl 35 könnte so beispielsweise durch fünf Kugeln der Wertigkeit 1 und drei Kugeln der Wertigkeit 10 dargestellt werden.

Der Datentyp `Real`

Zur Ausgabe von Daten des Typs `Real` wurden folgende Dialoge realisiert:

Segment-Ausgabe: Mit Hilfe dieses Dialoges kann sowohl ein Datum des Typs `Real` als auch ein Datum des Typs `Integer` ausgegeben werden. Dabei werden die jeweiligen Ziffern des Datums in einer 7-Segment-Darstellung abgebildet. Sie erscheinen rot gefärbt in einer Canvas mit schwarzem Hintergrund.

Säulen-Ausgabe: Dieser Dialog besteht im wesentlichen aus einer Canvas, auf der eine Art Reagenzglas dargestellt wird. Der Benutzer übergibt der entsprechenden Ausgabefunktion zwei Parameter, die als untere bzw. obere Grenze am unteren bzw. oberen Rand des Reagenzglases erscheinen. Die auszugebende Zahl, die innerhalb dieser Grenzen liegen muß, wird ebenfalls als Markierung dieses Reagenzglases dargestellt. Sie legt gleichzeitig die Markierung fest, wie hoch das Reagenzglas mit einer roten Flüssigkeit gefüllt ist.

Der Datentyp `String`

Folgende Dialoge dienen zur Ausgabe eines Wertes vom Typ `String`:

File-Ausgabe: Dieser Dialog verlangt als Ausgabeparameter den absoluten Pfad einer Textdatei und gibt den Inhalt dieser Datei in einer *Textfläche* (Widget, welches eine oder mehrere Textzeilen darstellen kann) aus. Rollbalken sorgen dafür, daß der Benutzer den sichtbaren Ausschnitt verändern kann. Wird ein ungültiger Dateiname angegeben, so bleibt die Textfläche leer.

Message-Ausgabe: Um dem Benutzer eine Nachricht zu übermitteln, kann dieser Dialog benutzt werden. Die entsprechende Meldung wird ähnlich der oben beschriebenen File-Ausgabe in einer Textfläche ausgegeben. Die Größe der Textfläche ist jedoch wesentlich kleiner. Eine große Schrift sorgt dafür, daß die Nachricht schnell und gut gelesen werden kann.

Der Datentyp `Boolean`

Zur Ausgabe eines Booleschen Wertes wurden folgende Dialoge realisiert:

Ampel-Ausgabe: Mittels einer Canvas wird in diesem Dialog eine Art Fußgänger-Ampel dargestellt, die genau zwei Zustände (rot und grün) besitzt, die als kleine runde Flächen in den entsprechenden Farben dargestellt sind. Je nach Wert des übergebenen Datums, erscheint eine dieser Flächen in einem helleren Farbton, während die jeweils andere leicht verdunkelt dargestellt wird. Dabei entspricht der Wert `#true` einem „Aufleuchten“ der grünen Fläche und der Wert `#false` einem „Aufleuchten“ der roten Fläche.

Smiley-Ausgabe: Ähnlich wie bei der oben beschriebenen Ampel-Ausgabe benutzt dieser Dialog zwei ausgezeichnete Grafiken zur Darstellung der Booleschen Werte `#true` und `#false`. Bei einer Ausgabe von `#true` erscheint ein lachendes Gesicht, während bei einer Ausgabe von `#false` ein trauriges Gesicht erscheint.

5.3.2 Ausgabe von ProSet-Werten komplexen Typs

Im folgenden sollen nun graphische Ausgabe-Dialoge für die komplexen ProSet-Typen `set` und `tuple` beschrieben werden. Dabei dienen sämtliche folgenden Dialoge sowohl zur Ausgabe eines Tupels als auch zur Ausgabe einer Menge.

Die Datentypen `set` und `tuple`

Grafik-Ausgabe: Mit Hilfe dieses Dialoges kann das auszugebende Datum graphisch als Baum dargestellt werden. Diese Darstellung entspricht im wesentlichen der in Kapitel 5.2.2 (Grafik-Eingabe) beschriebenen Form, verfügt jedoch über weitere Funktionen. Zu Anfang besteht die ausgegebene Grafik lediglich aus einem einzigen Symbol, welches je nach Typ des ausgegebenen Datums (Tupel oder Menge) variiert. Mit Hilfe der Maus (Doppelklick der linken Maustaste) kann der Benutzer die Darstellung der Graphik verändern. Dabei wird das doppelt angeklickte Symbol, sofern es sich um ein Tupel- bzw. Mengensymbol handelt, baumartig mit den jeweiligen Elementen als Söhne dargestellt. Durch doppeltes Anklicken der Wurzel eines Teilbaumes mit der mittleren Maustaste werden sämtliche dargestellten Söhne dieser Wurzel wieder entfernt. Übrig bleibt das entsprechende Mengen- bzw. Tupelsymbol der Wurzel.

Diagramm-Ausgabe: Dieser Dialog dient zur Ausgabe einer Menge bzw. eines Tupels von positiven Elementen ganzzahligen Typs. Dabei kann der Benutzer durch Drücken zweier ausgezeichneter Buttons die jeweilige Darstellung verändern. Die beiden Buttons tragen folgende Bezeichnungen:

- **Torte:** Durch Drücken dieses Buttons erzeugt der Benutzer eine Darstellung des ausgegebenen Datums als Tortendiagramm. Dabei erscheint jedes Element des Datums als farbiges Kreissegment. Die entsprechenden Werte erscheinen jeweils direkt neben den zugehörigen Kreisausschnitten. Handelt es sich bei dem ausgegebenen Datum um ein Tupel, so wird zusätzlich der Index des Elementes in Klammern hinter dem jeweiligen Wert angezeigt.
- **Balken:** Mit Hilfe dieses Buttons erzeugt der Benutzer eine Darstellung des ausgegebenen Datums als Balkendiagramm. Dabei erscheint jedes Element des Datums als farbiger Balken. Der zugehörige Wert erscheint jeweils über dem zugehörigen Balken. Handelt es sich bei dem ausgegebenen Datum um ein Tupel, so erscheint der Index der Elemente jeweils unter den sie repräsentierenden Balken.

Element-Ausgabe: Mit Hilfe dieses Dialoges kann der Benutzer eine Menge bzw. ein Tupel ausgeben. Dabei erscheinen die jeweiligen Elemente des ausgegebenen Datums in einer dafür vorgesehenen Listbox. Die Reihenfolge der Elemente in der Listbox entspricht dabei der Reihenfolge der Elemente im ausgegebenen Datum.

Filter-Ausgabe: Dieser Dialog entspricht im wesentlichen dem zuletzt beschriebenen Dialog zur Ausgabe eines Datums des Typs `set` bzw. `tuple`. Zusätzlich kann der Benutzer mit Hilfe eines Textfeldes einen Filter angeben, der die Menge der Elemente des ausgegebenen Datums filtert. Die Syntax des Filters entspricht der in Kapitel 5.2.1 angegebenen Syntax (`Filename-Eingabe`) und soll hier nicht wiederholt beschrieben werden.

Kapitel 6

Die ProSet-Bibliothek

Dieses Kapitel beschäftigt sich mit dem Entwurf der ProSet-Bibliothek. In Kapitel 4.6 wurde bereits erwähnt, daß die Komponenten der ProSet-Bibliothek im wesentlichen ProSet-spezifische Hüllen für die Komponenten der C-Bibliothek bilden. Kapitel 6.1 beschreibt zunächst die gemeinsame Struktur der verschiedenen ProSet-Bibliothekskomponenten. Kapitel 6.2 geht dann genauer auf die einzelnen Komponenten der ProSet-Bibliothek ein.

6.1 Struktur der Bibliothekskomponenten

Dieses Kapitel beschäftigt sich mit der Struktur der ProSet-Bibliothekskomponenten. Dabei wollen wir zunächst Komponenten zur Eingabe eines ProSet-Wertes betrachten. Sie besitzen eine Struktur, die folgendermaßen beschrieben werden kann:

```
procedure PS_XGet(<par_1>,<par_2>,...,<par_n>);
begin
  < Überprüfung der Parameter >
  < Transformationen in Parameter der C-Schnittstelle >
  < Aufruf der entsprechenden Funktion der C-Bibliothek über die
    C-Schnittstelle von ProSet >
  < Eventuelle Erzeugung einer ProSet-Ausnahme >
  < Rücktransformation des eingegebenen Datums >
 return Datum;
end of PS_XGet;
```

Zunächst werden sämtliche der Funktion übergebenen Parameter (u.a. Darstellungsattribute) sowohl auf ihren Typ als auch auf ihren Wert hin überprüft. Erfüllt einer dieser Parameter die geforderten Eigenschaften nicht, so wird eine ProSet-Ausnahme erzeugt (`xio_type_mismatch()` bei Typfehlern bzw. `xillegal_operand()` bei Wertfehlern). Nachdem sämtliche Parameter auf ihre Korrektheit hin überprüft worden sind, werden sie, wenn nötig, in eine Darstellung transformiert, deren Typ von der C-Schnittstelle von ProSet unterstützt wird (Die C-Schnittstelle von ProSet unterstützt bisher nur primitive C-Datentypen). Hierauf gehen wir genauer in Kapitel 7 ein.

Nach Transformation der Parameter erfolgt schließlich der Aufruf der zugehörigen Funktion der C-Bibliothek (siehe Kapitel 4.6). Nach Beendigung des so erzeugten Eingabe-Dialoges wird der zurückgelieferte Wert darauf hin überprüft, ob eine ProSet-Ausnahme erzeugt werden muß. In diesem Fall handelt es sich bei dem zurückgelieferten Wert um einen ausgezeichneten Wert. Beispielsweise wird der Wert `Cancel` zurückgeliefert, falls der Benutzer die Eingabeaufforderung mit Hilfe des `Abbruch`-Buttons abgebrochen hat. Wird keine ProSet-Ausnahme erzeugt, so wird das eingegebene Datum, welches ebenfalls eine schnittstellenverträgliche Darstellung besitzt, in ein zugehöriges ProSet-Datum umgewandelt, welches schließlich als Rückgabewert der Funktion `PS_XGet` zurückgeliefert wird.

Komponenten zur Ausgabe eines ProSet-Wertes besitzen eine Struktur, die folgendermaßen beschrieben werden kann:

```
procedure PS_XPut(Datum,<par_2>,...,<par_n>);
begin
  < Überprüfung der Parameter >
  < Transformationen in Parameter der C-Schnittstelle >
  < Aufruf der entsprechenden Funktion der C-Bibliothek über die
    C-Schnittstelle von ProSet >
  < Eventuelle Erzeugung einer ProSet-Ausnahme >
end of PS_XPut;
```

Die Struktur dieser Komponenten entspricht im wesentlichen der Struktur der oben beschriebenen Komponenten zur Eingabe eines ProSet-Wertes. Anhand von Statusmeldungen, die nach Aufruf der Ausgabefunktionen der C-Bibliothek zurückgeliefert werden, wird entschieden, ob eine ProSet-Ausnahme erzeugt werden muß oder nicht. Die Funktion `PS_XPut` liefert selbst keinen Wert zurück.

6.2 Komponenten der ProSet-Bibliothek

In diesem Kapitel erfolgt nun eine Auflistung von Beschreibungen sämtlicher Komponenten der zu realisierenden ProSet-Bibliothek. Dabei enthält eine Beschreibung den Namen, die zugehörigen Parameter, den Anwendungsbereich, das zugehörige Formular (vgl. Kapitel 5) und die zugehörige Funktion der C-Bibliothek. Zudem erfolgt ein ProSet-Beispielaufruf am Ende jeder Beschreibung. Wir beginnen mit den Komponenten zur Eingabe von ProSet-Werten.

6.2.1 Eingabe von ProSet-Werten

Name: `PS_XGet`

Parameter: *keine*

Aufgabe: Eingabe beliebiger ProSet-Werte

Formular: `Standard-Eingabe`

C-Funktion: `XGet()`

Beispiel: `Datum := PS_XGet();`

Name: `PS_XGetGetMap`

Parameter: Zwei nicht-leere Mengen `Set1` und `Set2`

Aufgabe: Eingabe einer `Map` mit `Set1` als Vorbereich und `Set2` als Nachbereich

Formular: `Map-Eingabe`

C-Funktion: `XGetMap()`

Beispiel: `Datum := PS_XGetMap({1,2,3},{4,5,6});`

Name: `PS_XGetReg`

Parameter: Ein regulärer Ausdruck `RegExp` vom Typ `string`

Aufgabe: Eingabe beliebiger ProSet-Werte, die durch den regulären Ausdruck `RegExp` beschrieben werden.

Formular: `Regular-Eingabe`

C-Funktion: `XGetReg()`

Beispiel: `Datum := PS_XGetReg("^[0-9]*$");`

Mit Hilfe dieser Funktion wurden entsprechend der ProSet-Datentypen sechs Funktionen zur Eingabe eines ProSet-Wertes realisiert, indem die aufzurufende C-Funktion mit einem entsprechenden regulären Ausdruck parametrisiert wurde. Die regulären Ausdrücke beschreiben jeweils sämtliche Daten des jeweiligen ProSet-Typs, so daß nur Daten dieses Typs akzeptiert werden. Die Namen der Komponenten lauten:

- `PS_XGetInt`
- `PS_XGetReal`
- `PS_XGetBoolean`
- `PS_XGetString`
- `PS_XGetTuple`
- `PS_XGetSet`

Parameter: *keine*

Aufgabe: Eingabe getypter ProSet-Werte

Formular: `Regular-Eingabe`

C-Funktion: `XGetReg()`

Beispiel: `Datum := PS_XGetInt();`

Name: `PS_XGetIntBlock`

Parameter: *keine*

Aufgabe: Eingabe einer ganzen Zahl

Formular: `Integer-Zehner-Block-Eingabe`

C-Funktion: `XGetIntBl()`

Beispiel: `Datum := PS_XGetIntBlock();`

Name: `PS_XGetRealBlock`

Parameter: Zwei ganzzahlige, positive Werte `vor` und `nach`

Aufgabe: Eingabe einer rationalen Zahl mit `vor` Vorkommastellen und `nach` Nachkommastellen

Formular: `Real-Zehner-Block-Eingabe`

C-Funktion: `XGetRealBl()`

Beispiel: `Datum := PS_XGetRealBlock(3,2);`

Name: `PS_XGetIntScale`

Parameter: Zwei ganzzahlige Werte `oben` und `unten`, wobei `unten<oben`

Aufgabe: Eingabe einer ganzen Zahl zwischen `unten` und `oben`

Formular: `Schieberegler-Eingabe`

C-Funktion: `XGetIntSc()`

Beispiel: `Datum := PS_XGetIntScale(-100,100);`

Name: `PS_XGetTupSet`

Parameter: *keine*

Aufgabe: Eingabe beliebiger Mengen oder Tupel

Formular: `Grafik-Eingabe`

C-Funktion: `XGetTupSet()`

Beispiel: `Datum := PS_XGetTupSet();`

Name: `PS_XGetFilename`

Parameter: *keine*

Aufgabe: Eingabe eines Dateinamens mit absolutem Pfad

Formular: `Filename-Eingabe`

C-Funktion: `XGetFilename()`

Beispiel: `Datum := PS_XGetFilename();`

Name: `PS_XGetChoice`

Parameter: *keine*

Aufgabe: Eingabe beliebiger ProSet-Werte

Formular: `Wahl-Eingabe`

C-Funktion: `XGetChoice()`

Beispiel: `Datum := PS_XGetChoice();`

Name: `PS_XGetRealBruch`

Parameter: *keine*

Aufgabe: Eingabe einer rationalen Zahl

Formular: `Bruch-Eingabe`

C-Funktion: `XGetRealBr()`

Beispiel: `Datum := PS_XGetRealBruch();`

Name: `PS_XGetRadioBool`

Parameter: *keine*

Aufgabe: Eingabe eines Booleschen Wertes

Formular: `True-oder-False-Eingabe`

C-Funktion: `XGetRadioBool()`

Beispiel: `Datum := PS_XGetRadioBool();`

Name: `PS_XGetAnswer`

Parameter: `Frage` vom Typ `string`

Aufgabe: Beantwortung der `Frage` mit Ja oder Nein

Formular: `Antwort-Eingabe`

C-Funktion: `XGetAntwort()`

Beispiel: `Datum := PS_XGetAnswer("Sind Sie jung ?");`

Name: `PS_XGetRegSet`

Parameter: Ein regulärer Ausdruck `RegExp` vom Typ `string`

Aufgabe: Eingabe beliebiger Mengen, deren Elemente durch den regulären Ausdruck `RegExp` beschrieben werden.

Formular: `Regular-Set-Eingabe`

C-Funktion: `XGetRegSet()`

Beispiel: `Datum := PS_XGetRegSet("^[0-9]+$");`

Ähnlich wie bei `PS_XGetReg` werden auch hier entsprechend der sechs ProSet-Typen spezielle Komponenten gebildet, die zur Eingabe von getypten Mengen dienen. Sie lauten:

- `PS_XGetIntSet`
- `PS_XGetRealSet`
- `PS_XGetBooleanSet`
- `PS_XGetStringSet`
- `PS_XGetTupleSet`
- `PS_XGetSetSet`

Parameter: *keine*

Aufgabe: Eingabe getypter Mengen

Formular: `Regular-Set-Eingabe`

C-Funktion: `XGetRegSet()`

Beispiel: `Datum := PS_XGetIntSet();`

Name: `PS_XGetIntTuple`

Parameter: Ein positiver, ganzzahliger Wert `Anzahl`, sowie zwei ganzzahlige Werte `von` und `bis`, wobei `von<bis`

Aufgabe: Eingabe eines Tupels aus `Anzahl` ganzen Zahlen, die nicht kleiner als `von` und nicht größer als `bis` sind

Formular: `Equalizer-Eingabe`

C-Funktion: `XGetIntTupelEq()`

Beispiel: `Datum := PS_XGetIntTuple(5,-20,20);`

6.2.2 Ausgabe von ProSet-Werten

In diesem Kapitel werden nun Dialoge zur Ausgabe von ProSet-Werten vorgestellt. Die Notation erfolgt analog zu Kapitel 6.2.1

Name: `PS_XPut`

Parameter: `Ausgabedatum` (beliebiger Typ)

Aufgabe: Ausgabe beliebiger ProSet-Werte

Formular: `Standard-Ausgabe`

C-Funktion: `XPut()`

Beispiel: `PS_XPut("hallo");`

Name: `PS_XPutFile`

Parameter: `Ausgabedatum` (vom Typ `string`)

Aufgabe: Ausgabe eines Textfiles

Formular: `File-Ausgabe`

C-Funktion: `XPutFile()`

Beispiel: `PS_XPutFile("/home/diplom/puetter/mbox");`

Name: `PS_XPutIntAbakus`

Parameter: `Ausgabedatum` (vom Typ `integer`)

Aufgabe: Ausgabe einer ganzen Zahl

Formular: `Abakus-Ausgabe`

C-Funktion: `XPutIntAbakus()`

Beispiel: `PS_XPutIntAbakus(12000);`

Name: `PS_XPutSeg`

Parameter: `Ausgabedatum` (vom Typ `real` oder `integer`)

Aufgabe: Ausgabe einer Zahl beliebigen Typs

Formular: `Segment-Ausgabe`

C-Funktion: `XPutRealSeg()`

Beispiel: `PS_XPutSeg(-4.67);`

Name: `PS_XPutTupSetBr`

Parameter: `Ausgabedatum` (vom Typ `set` oder `tuple`)

Aufgabe: Ausgabe beliebiger Mengen oder Tupel

Formular: `Filter-Ausgabe`

C-Funktion: `XPutSetBr()`

Beispiel: `PS_XPutTupSetBr([1,2,3]);`

Name: `PS_XPutTupSetLb`

Parameter: `Ausgabedatum` (vom Typ `set` oder `tuple`)

Aufgabe: Ausgabe beliebiger Mengen oder Tupel

Formular: `Element-Ausgabe`

C-Funktion: `XPutSetLb()`

Beispiel: `PS_XPutTupSetLb({1,2,3});`

Name: `PS_XPutTorte`

Parameter: `Ausgabedatum` (vom Typ `set` oder `tuple` mit ganzzahligen, positiven Elementen)

Aufgabe: Ausgabe einer Menge bzw. eines Tupels, deren Elemente ganzzahlig und positiv sind.

Formular: `Diagramm-Ausgabe`

C-Funktion: `XPutTorte()`

Beispiel: `PS_XPutTorte([10,20,45,67]);`

Name: `PS_XPutTupSetGr`

Parameter: `Ausgabedatum` (vom Typ `set` oder `tuple`)

Aufgabe: Ausgabe beliebiger Mengen oder Tupel

Formular: `Grafik-Ausgabe`

C-Funktion: `XPutSetGr()`

Beispiel: `PS_XPutSetGr([1.2,4]);`

Name: `PS_XPutIntChoice`

Parameter: `Ausgabedatum` (vom Typ `integer` und positiv)

Aufgabe: Ausgabe ganzer und positiver Zahlen

Formular: `Wahl-Ausgabe`

C-Funktion: `XPutIntChoice()`

Beispiel: `PS_XPutIntChoice(56);`

Name: `PS_XPutMessage`

Parameter: `Ausgabedatum` (vom Typ `string`)

Aufgabe: Ausgabe einer Nachricht

Formular: `Message-Ausgabe`

C-Funktion: `XPutMessage()`

Beispiel: `PS_XPut("Berechnung ist abgeschlossen");`

Name: `PS_XPutSmiley`

Parameter: `Ausgabedatum` (vom Typ `boolean`)

Aufgabe: Ausgabe eines Booleschen Wertes

Formular: `Smiley-Ausgabe`

C-Funktion: `XPutSmiley()`

Beispiel: `PS_XPutSmiley(#true);`

Name: `PS_XPutAmpel`

Parameter: `Ausgabedatum` (vom Typ `boolean`)

Ausgabe: Ausgabe eines Booleschen Wertes

Formular: `Ampel-Ausgabe`

C-Funktion: `XPutAmpel()`

Beispiel: `PS_XPutAmpel(#false);`

Name: `PS_XPutRealThermo`

Parameter: `Ausgabedatum` (vom Typ `real`), `unten` (vom Typ `integer`), `oben` (vom Typ `integer`)

Aufgabe: Ausgabe einer rationalen Zahl, die größer als `unten` und kleiner als `oben` ist

Formular: `Säulen-Ausgabe`

C-Funktion: `XPutRealThermo()`

Beispiel: `PS_XPutRealThermo(3.23,-10,10);`

Kapitel 7

Implementierung

Dieses Kapitel beschäftigt sich mit der Implementierung graphischer Dialoge. Es soll zudem als Leitfaden für eine Erweiterung der ProSet-Bibliothek dienen (siehe Kapitel 7.12). Zunächst betrachten wir den Aufbau der C-Bibliothekskomponenten zur Ein- und Ausgabe beliebiger Werte. In Kapitel 7.2 gehen wir genauer auf die Implementierung der bereits in Kapitel 4.2.2 und 4.2.3 beschriebenen graphischen Ein- und Ausgabeprozesse ein. Die Implementierung weiterer Konzepte (Ausnahmebehandlung, Hilfesystem) wird in den darauffolgenden Kapiteln beschrieben. Kapitel 7.6 beschäftigt sich näher mit Tcl/Tk. Insbesondere werden dort anhand von Beispielen Tcl-Skripte zur Ein- und Ausgabe von ProSet-Werten beschrieben. Innerhalb dieser Skripten werden häufig bestimmte Funktionen genutzt, die zu Bibliotheken zusammengefaßt und in Kapitel 7.11 beschrieben sind. Kapitel 7.12 beschreibt abschließend die wesentlichen Schritte zur Erweiterung der ProSet-Bibliothek.

7.1 Die C-Bibliothek

In Kapitel 4.6 wurde erwähnt, daß die Komponenten der zu realisierenden C-Bibliothek eine Ein- und Ausgabe von beliebigen Daten basierend auf dem Konzept der internen Kontrolle ermöglichen. Im folgenden soll nun die grundlegende Struktur dieser Komponenten anhand von Beispielen erläutert werden. Dabei beginnen wir mit der Beschreibung einer C-Bibliotheks-Komponente zur Eingabe eines ProSet-Wertes (zugehöriges Formular: `Regular-Eingabe`). Eine Beschreibung besteht dabei aus Code-Fragmenten, deren Erklärung jeweils nach den entsprechenden Fragmenten erscheint.

7.1.1 Bibliothekskomponenten zur Eingabe von ProSet-Werten

```
char* XGetReg(Regular)
char** Regular;
```

Die Funktion `XGetReg` besitzt einen Parameter `Regular`, der als Darstellungsattribut dem Eingabeprozeß `Input` (siehe dazu Kapitel 7.2.1) als Parameter übergeben wird.

```
{
 char *Ergebnis;
```

```
int Pid;        /*Prozessidentifikator des erzeugten
                  Kindprozesses*/
char *Msg;      /*MessageQueue-Identifikator*/
char *Show;     /*Show-Parameter*/
char *Attribut; /*Darstellungsattribut*/
```

Sämtliche notwendigen Variablen sind nun deklariert. Einige von ihnen erscheinen später als Parameter des Eingabeprozesses `Input` (siehe Kapitel 7.2.1).

```
signal(SIGINT,Abbruch);
```

Hier wird der Signal-Handler für ein eintreffendes Abbruch-Signal, welches vom Benutzer erzeugt werden kann (`Ctrl-C`) festgelegt.

```
Msg = mymalloc(100); /*Speicherallokation mit
                       entsprechender Fehlerbehandlung*/
Show = mymalloc(100);
Attribut = (char*)Regular;
if (MsgQueueID == -1) MsgQueueID = msgget(IPC_PRIVATE,0640);
sprintf(Msg,"%d",MsgQueueID);
sprintf(Show,"%d",0);
```

Nun sind sämtliche Variablen, mit denen der Eingabeprozeß `Input` parametrisiert wird, initialisiert worden. Auf die Bedeutung der einzelnen Parameter gehen wir in Kapitel 7.2.1 und 7.2.2 noch genauer ein.

```
if (CheckPath() == 0)
{
 Ergebnis = mymalloc(strlen("NoPath")+1);
 sprintf(Ergebnis,"%s","NoPath");
 free(Msg);
 free(Show);
 return Ergebnis;
}
```

Hier wird mittels der Funktion `CheckPath()` überprüft, ob das *Modulefile* `prosetgui` geladen ist. Ein Modulefile enthält im wesentlichen Pfadbeschreibungen, die in sogenannten Environment-Variablen abgelegt werden. In der Regel benutzt man innerhalb eines Programmes dann anstelle von konstanten Pfaden diese Environment-Variablen. Ein Vorteil dieser Vorgehensweise liegt darin, daß mögliche Änderungen der Pfade nur in dem entsprechenden Modulefile getätigt werden müssen und nicht in sämtlichen Programmen, die die entsprechenden Pfade nutzen. Falls das Modulefile nicht geladen ist, wird der ausgezeichnete Wert `NoPath` zurückgeliefert und die Funktion `XGetReg` an dieser Stelle verlassen.

```
if (Pinpid == -1)
{
  SetPinboardID();
  Pinpid = fork();
}
```

Falls nicht zuvor bereits ein Pinboard gestartet wurde (`Pinpid==-1`), wird hier mittels des Unix-Systemaufrufes `fork` ein Kindprozeß für das Pinboard erzeugt. Der Aufruf `SetPinboardID()` erzeugt einen Pinboard-Identifikator, der als Parameter dem Eingabeprozeß `Input` übergeben wird (siehe dazu 7.2.1). Der Systemaufruf `fork` liefert für den Vaterprozeß den Prozeß-Identifikator des Kindprozesses zurück. Für den Kindprozeß ist dieser Identifikator 0. Anhand dieses Identifikators kann nun entschieden werden, welchen Code der Kindprozeß ausführen soll.

```
if (Pinpid == 0) /*Code des Kindprozesses*/
  {
   StartPinboard();
  }
```

Der durch `fork` erzeugte Kindprozeß startet mit Hilfe der Anweisung `StartPinboard()` ein Pinboard.

```
 else /*Code des Vaterprozesses*/
 {
  pid = fork();
```

Der Vaterprozeß erzeugt mit Hilfe von `fork` einen Kindprozeß.

```
  if (pid == 0) /*Code des Kindprozesses*/
  {
   char* Pfad;
   Pfad = mymalloc(strlen(getenv("INP"))+strlen("/Input")+1);
   sprintf(Pfad,"%s/Input",getenv("INP"));
   execl(Pfad,"Input",PinboardID,Msg,Show,"RegularEingabe",
         Attribut,NULL);
  }
```

Der Kindprozeß sorgt dafür, daß das entsprechende Dialogfenster der Eingabeaufforderung `Regular-Eingabe` auf dem Bildschirm erscheint, indem es mit Hilfe des Unix-Systemaufrufes `execl` den Eingabeprozeß `Input` mit den zugehörigen Parametern startet.

```
  else /*Code des Vaterprozess*/
  {
   free(Msg);
   free(Show);
   BMessage.mtype = 0;
   while (BMessage.mtype == 0)
   {
    ReturnVal = msgrcv(MsgQueueID,&BMessage,250,-5,0);
   }
```

Nachdem sämtliche Prozesse erzeugt wurden, wartet der Vaterprozeß durch Aufruf von `msgrcv` (siehe dazu [HS87]) auf eine Nachricht des Kindprozesses. Zuvor wird jedoch der oben angeforderte Speicher wieder freigegeben.

```
   switch (BMessage.mtype) {
   case 1: Ergebnis = mymalloc(250);
           strcpy(Ergebnis,&(BMessage.Wert[0]));
           return Ergebnis;break;
   case 2: Ergebnis = mymalloc(strlen("NoDisplay")+1);
           sprintf(Ergebnis,"%s","NoDisplay");
           return Ergebnis;break;
   case 3: Ergebnis = mymalloc(strlen("Close")+1);
           sprintf(Ergebnis,"%s","Close");
           return Ergebnis;break;
   case 4: Ergebnis = mymalloc(strlen("Cancel")+1);
           sprintf(Ergebnis,"%s","Cancel");
           return Ergebnis;break;
   default:
   }
  }
 }
}
```

Nachdem der Vaterprozeß eine Nachricht empfangen hat, wird sie anhand ihres Typs überprüft und es werden entsprechende Werte als Funktionsergebnis der Funktion `XGetReg` zurückgeliefert. Ist die eintreffende Nachricht beispielsweise vom Typ `1`, so wird der Nachrichteninhalt als korrektes ProSet-Datum interpretiert und als Funktionsergebnis in Form einer Zeichenkette zurückgeliefert. Für die Nachrichtentypen `2`, `3` und `4` werden ausgezeichnete Werte zur Erzeugung einer ProSet-Ausnahme zurückgeliefert (siehe dazu Kapitel 7.5).

7.1.2 Bibliothekskomponenten zur Ausgabe von ProSet-Werten

Nachdem wir die Struktur der Komponenten zur Eingabe eines ProSet-Wertes beschrieben haben, folgt nun eine Beschreibung der Komponenten zur Ausgabe eines ProSet-Wertes (Säulen-Ausgabe).

```
int XPutSaeule(Datum,Unten,Oben)
char* Datum;
int Unten,Oben;
```

Die Ausgabefunktion vom Namen `XPutSaeule` besitzt drei Parameter. `Datum` enthält das auszugebende ProSet-Datum in transformierter Darstellung (als Zeichenkette). `Unten` und `Oben` sind Darstellungsattribute (obere und untere Grenze der darzustellenden Säule).

```
{
 int pid;
 int  *Status;       /*Rueckgabewert der Ausgabefunktion*/
 char *Vaterpid;     /*Prozessidentifikator des
                       Vaterprozesses*/
 char *Show;         /*Show-Parameter*/
 char *Ausgabedatum; /*auszugebendes Datum*/
 char *UntereGrenze; /*untere Grenze*/
 char *ObereGrenze;  /*obere Grenze*/
```

Zunächst werden sämtliche notwendigen Variablen deklariert. Einige von ihnen dienen als Parameter des Ausgabeprozesses `Output`.

```
pid = getpid();
```

`getpid` liefert den Prozeß-Identifikator des Prozesses, der `getpid()` aufruft, zurück. Dieser Prozeßidentifikator wird später zur Signalübertragung vom Kindprozeß zum Vaterprozeß benötigt.

```
Vaterpid = mymalloc(100);
Show = mymalloc(100);
Ausgabedatum = (char*)Datum;
UntereGrenze = mymalloc(100);
ObereGrenze = mymalloc(100);
sprintf(Vaterpid,"%d",pid);
sprintf(Show,"%d",0);
sprintf(UntereGrenze,"%d",Unten);
sprintf(ObereGrenze,"%d",Oben);
```

Sämtliche Variablen sind nun mit den nötigen Werten belegt. Auf die Bedeutung der Parameter gehen wir in Kapitel 7.2.2 noch genauer ein.

```
signal(SIGINT,Abbruch);
signal(SIGUSR2,Fertig);
signal(SIGUSR1,NoDisplay);
```

Hier werden die Signal-Handler für das Abbruch-Signal, sowie für zwei benutzerdefinierte Signale festgelegt. Benutzerdefinierte Signale werden unter anderem zur Propagation von Ausnahmezuständen benötigt (siehe dazu auch Kap. 7.5).

```
if (CheckPath() == 0)
{
 free(Vaterpid);
 free(Show);
 free(UntereGrenze);
 free(ObereGrenze);
 return 2;
}
```

Für den Fall, daß das Modulefile `prosetgui` nicht geladen ist, wird der Statuswert `2` als Funktionsergebnis zurückgeliefert und die Funktion `XPutSaeule` an dieser Stelle verlassen.

```
if (Pinpid == -1)
{
 SetPinboardID();
 Pinpid = fork();
}
if (Pinpid == 0) /*Code des Kindprozesses*/
```

```
{
  StartPinboard();
}
else /*Code des Vaterprozesses*/
{
 pid = fork();
 if (pid == 0) /*Code des Kindprozesses*/
 {
  char* Pfad;
  Pfad = mymalloc(strlen(getenv("OUT"))+strlen("/Output")+1);
  sprintf(Pfad,"%s/Output",getenv("OUT"));
  execl(Pfad,"Output",PinboardID,Vaterpid,Show,Ausgabedatum,
        "RealThermoAusgabe",UntereGrenze,
        ObereGrenze,NULL);
 }
```

Dieser Programmteil entspricht im wesentlichen dem Code in Kapitel 7.1.1 zur Erzeugung des Pinboards und des entsprechenden Dialogfensters. Er soll hier nicht wiederholt beschrieben werden.

```
 else /*Code des Vaterprozesses*/
 {
  ret = 0;
  while (OK == 0) {}
  if (OK == 2) {ret = 1;}
  OK = 0;
  free(Vaterpid);
  free(Show);
  free(UntereGrenze);
  free(ObereGrenze);
  return ret;
 }
 }
}
```

`OK` bezeichnet eine globale Variable, die von zwei Signalhandlern, die auf die benutzerdefinierten Signale `SIGUSR1` und `SIGUSR2` reagieren, belegt wird. Anhand des Wertes von `OK` erkennt der Vaterprozeß, ob es zu Ausnahmezuständen in dem Ausgabeprozeß `Output` gekommen ist. Falls `OK` auf den Wert `2` gesetzt wird, wird der Statuswert `1` zurückgeliefert, um den entsprechenden Fehlerzustand in das ProSet-Programm zu propagieren, wo eine entsprechende Fehlerbehandlung stattfinden kann. Schließlich wird `OK` wieder auf den Wert 0 zurückgesetzt, um bei folgenden Ausgaben keine fehlerhaften Rückgabewerte zu erzeugen. Verläuft die Ausgabe ohne Fehler, wird der Statuswert 0 zurückgeliefert.

Der Benutzer sollte nun in der Lage sein, eigene C-Bibliothekskomponenten zu konstruieren, da sich die einzelnen Komponenten nur in wenigen Punkten unterscheiden. Kernstücke dieser Komponenten sind die Anweisungen zum Starten der graphischen Ein- bzw. Ausgabeprozesse, die im folgenden Kapitel näher erläutert werden.

7.2 Graphische Prozesse

Im folgenden sollen nun die Prozesse `Input` und `Output` zur graphischen Ein- und Ausgabe von beliebigen Werten und deren Parameter beschrieben werden. Dabei handelt es sich um die von den Funktionen der C-Bibliothek erzeugten Kindprozesse, die dafür sorgen, daß ein entsprechendes Dialogfenster zur Ein- bzw. Ausgabe von beliebigen Werten auf dem Bildschirm erscheint. Um die bestehende C-Bibliothek erweitern zu können, muß der Benutzer Gebrauch von diesen Prozessen machen. Dabei muß darauf geachtet werden, daß diese korrekt parametrisiert werden.

7.2.1 Der Eingabeprozeß `Input`

Der Eingabeprozeß `Input` kann innerhalb einer Shell folgendermaßen aufgerufen werden:

```
Input PinID MqId Show Skript Arg1 ... Argn
```

Die Parameter des Eingabeprozesses `Input` erscheinen zusammen mit `Input` als Parameter des Unix-Systemaufrufes `execl` und haben im einzelnen die folgende Bedeutung:

PinID: Identifikator des Pinboards, der benötigt wird, um mehrere ProSet-Programme mit graphischer Ein- und Ausgabe parallel zu starten, ohne daß es dabei zu Konfliktsituationen kommt.

MqID: Identifikator der MessageQueue, über die eine Inter-Prozeß-Kommunikation mit dem Vaterprozeß stattfindet. Er dient zum Zurücksenden des eingegebenen Datums.

Show: Parameter, der bis in das Tcl-Skript durchgereicht wird. Er gibt an, ob das zu erzeugende Dialogfenster das Aussehen eines reproduzierten Dialogfensters besitzen soll oder nicht. Bei Aufruf innerhalb der C-Bibliothek ist dieser Parameter immer gleich 0. Aus dem Pinboard heraus wird der entsprechende Prozeß mit dem Parameter 1 aufgerufen.

Skript: Name des Tcl-Skriptes, welches zur Laufzeit von `Input` interpretiert wird. Es legt Aussehen und Verhalten des Dialoges fest.

Arg1, ..., Argn: Hierbei handelt es sich um Darstellungsattribute, mit denen das zugehörige Skript parametrisiert wird. Während es sich bei dem Parameter `Show` um ein obligatorisches Darstellungsattribut handelt, ist der Gebrauch von `Arg1`, ..., `Argn` optional. Sie hängen von der Art des benutzten Dialoges ab. Die Darstellungsattribute `Arg1`, ..., `Argn` werden bis in das zugehörige Tcl-Skript durchgereicht und sind dort als `Argv1`, ..., `Argvn` bekannt.

Weiterhin verfügt der Eingabeprozeß `Input` über folgende Eigenschaften:

- Wird der Eingabeprozeß über die Menüleiste (Eintrag: `Close`) beendet, so wird anstelle des eingegebenen Datums eine speziell dafür vorgesehene Nachricht an den Vaterprozeß gesendet. Die aktuelle Eingabefunktion der C-Bibliothek reagiert auf diese Nachricht durch Rückgabe des ausgezeichneten Wertes `Close`, um eine Ausnahmebehandlung anzustoßen. Der Wert `Close` wird nicht als eingegebener ProSet-Wert interpretiert, da er keinem ProSet-Typen entspricht.

- Ist die Umgebungsvariable `DISPLAY` nicht gesetzt, reagiert der Eingabeprozeß ebenfalls durch Verschickung einer speziell dafür vorgesehenen Nachricht, um damit eine Ausnahmebehandlung einzuleiten. Die aktuelle Eingabefunktion der C-Bibliothek liefert hierfür den Wert `NoDisplay`.
- Zwischen dem Eingabeprozeß `Input` und dem zu interpretierenden Tcl-Skript besteht eine Verbindung in Form eines *Variablen-Links* (siehe dazu [Ous94, Abschnitt 34.5]). Dieser Link dient dazu, Variablen innerhalb eines Tcl-Skript in der darunter liegenden C-Applikation bekannt zu machen. Weiterhin werden eine Anzahl von C-Variablen (`Show`, `PinID`, ...) in dem zugehörigen Tcl-Skript bekannt gemacht. Hierbei handelt es sich jedoch nicht um die eben beschriebenen Variablen-Links.
- Der Eingabeprozeß `Input` erzeugt bei seiner Ausführung ein Tcl-Kommando namens `Sende`. Dieses Kommando kann in dem zugehörigen Tcl-Skript zusammen mit einem Parameter `Typ` aufgerufen werden. Bei Ausführung dieses Kommandos wird der in der Tcl-Variablen `Input` gespeicherte Wert als Nachrichteninhalt zusammen mit `Typ` als Nachrichtentyp an den Elternprozeß gesendet.

Die Abbildung 7.1 soll diesen Sachverhalt nochmals verdeutlichen.

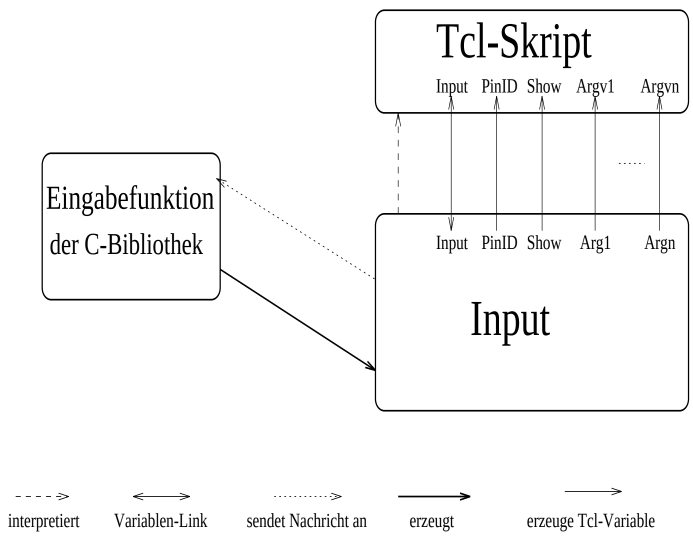


Abbildung 7.1: Der Eingabeprozeß `Input`

7.2.2 Der Ausgabeprozeß `Output`

Der Ausgabeprozeß `Output` kann innerhalb einer Shell folgendermaßen aufgerufen werden:

```
Output PinID VPid Show Datum Skript Arg1 ... Argn
```

Die Parameter haben dabei folgende Bedeutung:

PinID: siehe `Input`

VPid: Prozeß-ID des Vaterprozesses, der benötigt wird, um eine Signalkommunikation mit dem Vaterprozeß zu ermöglichen.

Show: siehe `Input`

Datum: Hierbei handelt es sich um das auszugebende Datum als Zeichenkette.

Skript: siehe `Input`

Arg1, ..., Argn: siehe `Input`

Genau wie der oben beschriebene Eingabeprozeß `Input` besitzt der Ausgabeprozeß `Output` vier obligatorische Parameter und beliebig viele optionale Parameter. Zur Kommunikation mit dem Vaterprozeß benutzt der Ausgabeprozeß Unix-Signale (siehe dazu [HS87]). In folgenden Fällen findet eine solche Kommunikation statt:

- Falls die Umgebungsvariable `DISPLAY` nicht gesetzt ist, wird das benutzerdefinierte Signal `SIGUSR1` an den Vaterprozeß gesendet. Die entsprechende Ausgabefunktion liefert in diesem Fall einen Statuswert zurück und stößt damit im ProSet-Programm eine Ausnahmebehandlung an.
- Auch der Ausgabeprozeß `Output` verfügt über einen Variablen-Link, der das auszugebende Datum in dem zugehörigen Tcl-Skript als Variable namens `Output` bekannt macht. Zudem wird analog zum Eingabeprozeß `Input` eine Anzahl von Variablen in dem zugehörigen Tcl-Skript bekannt gemacht.
- Ein applikationseigenes Tcl-Kommando namens `SetzeFertig` wird genau dann aufgerufen, falls kein Fehlerzustand eingetreten ist. Durch den Aufruf dieses Kommandos wird das benutzerdefinierte Signal `SIGUSR2` an den Vaterprozeß gesendet.

Die Abbildung 7.2 soll diesen Sachverhalt nochmals verdeutlichen.

7.3 Das Hilfesystem

Laut den Anforderungen in Kapitel 3 sollte für Eingabe-Dialoge ein *Hilfesystem* zur Verfügung gestellt werden. Unter einem Hilfesystem wollen wir einen Mechanismus verstehen, der den Benutzer zu jedem Zeitpunkt mit kontextsensitiven Informationen über das Aussehen und Verhalten, sowie über die Bedienung einer Applikation und ihrer Teile versorgt. Die Informationsversorgung soll hier mit Hilfe von Texten, die in einem Dialogfenster abgebildet werden, geschehen. Das zu realisierende Hilfesystem soll bezüglich eines Eingabe-Dialoges folgende Informationen zur Verfügung stellen:

- Welche Daten kann der Benutzer mit Hilfe des aktuellen Eingabe-Dialoges eingeben? (Typ des Eingabedatums)
- Wie ist das Dialogfenster des Eingabe-Dialoges zu bedienen?

An das Hilfesystem wurden ursprünglich folgende Anforderungen gestellt:

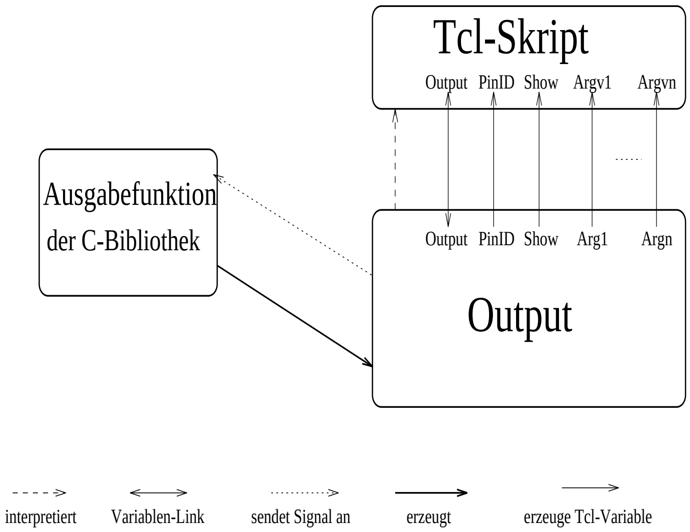


Abbildung 7.2: Der Ausgabeprozeß `Output`

- Die erklärenden Hilfetexte sollen nicht in den jeweiligen Eingabe-Dialogen gekapselt sein, sondern extern als Textdatei zur Verfügung stehen, um eine möglicherweise notwendige Änderung oder Erweiterung der Texte zu vereinfachen.

- Das Hilfesystem soll sowohl deutschsprachige als auch englischsprachige Benutzer ansprechen, indem die Hilfetexte wahlweise in englisch oder deutsch dargestellt werden können.

Zusätzlich verfügt das von uns realisierte Hilfesystem über Hypertext-Eigenschaften. Mit Hilfe der Maus kann der Benutzer speziell gekennzeichnete Wörter des Hilfetextes anklicken und somit den aktuell dargestellten Text abhängig vom angeklickten Wort verändern. In der Regel erscheint dann ein Text, der nähere Informationen zu dem angeklickten Wort enthält. Eine *History*-Funktion sorgt dafür, daß der Benutzer jederzeit zu dem ursprünglichen Text zurückkehren kann. Da sämtliche Texte in einer gemeinsamen Textdatei abgespeichert werden, muß diese Datei entsprechend *partitioniert* (unterteilt) werden. Dies geschieht mit Hilfe von *Marken*. Zur Partitionierung von Hilfetexten stellt das Hilfesystem folgende Marken zur Verfügung:

begin<*Marke*.ger>

bezeichnet den Anfang eines Hilfetextes zu einem Eingabe-Dialog in deutscher Sprache (Endung: `.ger`).

end<*Marke*.ger>

bezeichnet das Ende eines Hilfetextes in deutscher Sprache. Marken, die einen englischen Hilfetext eingrenzen, besitzen die Endung `.eng`. Innerhalb eines Hilfetextes können sogenannte

Hypertextmarken spezifiziert werden. Diese Hypertextmarken erscheinen später im Dialogfenster des Hilfesystems als hervorgehobene Wörter, die der Benutzer mit Hilfe der Maus anklicken kann. Hypertextmarken besitzen folgende Syntax:

`<hyp>`(*Begriff*,*Marke*)

In dem ausgegebenen Hilfetext wird dieser Ausdruck durch den Ausdruck *Begriff* ersetzt und farblich hervorgehoben. Klickt der Benutzer diesen Ausdruck mit Hilfe der Maus an, so wird der aktuell dargestellte Hilfetext durch den Hilfetext der zwischen den Marken `begin<`*Marke*`.ger>` und `end<`*Marke*`.ger>` steht, ersetzt, falls der deutschsprachige Modus aktiviert ist. Entsprechendes gilt für den englischsprachigen Modus.

Um einen Hilfedialog zu öffnen (dies geschieht, indem der Benutzer den `Hilfe`-Button eines Eingabe-Dialoges drückt), wurde eine Tcl-Prozedur namens `Help` implementiert. Der Aufruf innerhalb eines Tcl-Skriptes geschieht wie folgt:

`Help Marke arg1 ... argn`

Nach Ausführung eines `Help`-Kommandos erscheint ein Dialogfenster, welches den Text zwischen den Marken `begin<Marke.ger>` und `end<Marke.ger>` in der Datei `PSXio.hlp` darstellt (im deutschsprachigen Modus).

Neben einer Marke können zusätzlich Argumente übergeben werden, die in der Hilfetextdatei als `<arg1> ... <argn>` erscheinen können. In dem Dialogfenster des Hilfesystems werden diese Makros dann durch die konkreten Werte von `arg1 ... argn` ersetzt und farblich hervorgehoben. Das Dialogfenster, welches durch das oben beschriebene Kommando erzeugt wird, enthält neben einem `Ok`-Button zur Beendigung des Dialoges drei ausgezeichnete Buttons, die im folgenden näher beschrieben werden:

Zurück: Mit Hilfe dieses Buttons gelangt der Benutzer zu dem Hilfetext zurück, der dargestellt wurde, bevor der Benutzer die letzte Hypertextmarke angeklickt hat.

Englisch: Mit Hilfe dieses Buttons wechselt der Benutzer in den englischen Sprachmodus. Ab sofort werden sämtliche Hilfetexte in englischer Sprache dargestellt.

Deutsch: Mit Hilfe dieses Buttons wechselt der Benutzer in den deutschen Modus.

Folgendes Beispiel soll die Funktionsweise des Hilfesystems verdeutlichen. Dabei betrachten wir folgende Hifedatei:

```
begin<RegEingabe.ger>
Sie haben nun die Gelegenheit, ein <hyp>(ProSet-Datum,PSDatum) einzugeben,
welches den <hyp>(regulaeren Ausdruck,RegExp) <arg1> erfuellt.
Durch Druecken des Ok-Buttons beenden Sie Ihre Eingabe.
Durch Druecken des Abbruch-Buttons brechen Sie die Eingabe ab.
end<RegEingabe.ger>

begin<PSDatum.ger>
end<PSDatum.ger>
```

```
begin<RegExp.ger>
end<RegExp.ger>
```

Nach Aufruf von `Help RegEingabe "^[0-9]*$"` erscheint das in Abbildung 7.3 dargestellte Hilfefenster:

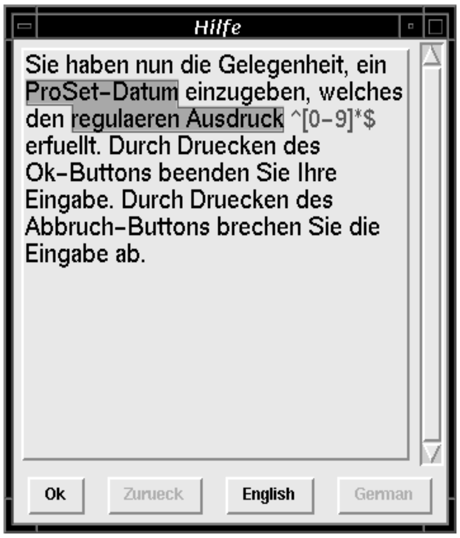


Abbildung 7.3: Hilfefenster mit erklärendem Text

7.4 Begrenzung der Ausgabefenster

Laut den Anforderungen in Kapitel 3 soll die Anzahl der gleichzeitig dargestellten Dialogfenster zur Ausgabe eines ProSet-Wertes begrenzt sein. Mit Hilfe des Pinboards kann der Benutzer jederzeit die maximale Anzahl von Ausgabefenstern festlegen. Sobald die Anzahl der dargestellten Ausgabefenster die vom Benutzer angegebene Anzahl überschreitet, wird das älteste von sämtlichen aktuellen, nicht reproduzierten Ausgabefenstern zerstört (FIFO-Verhalten). Dazu verwaltet das Pinboard eine Liste von Prozeß-Identifikatoren (pids), die mit jeder neuen Ausgabe aktualisiert wird. Die Anzahl der Elemente dieser Liste beträgt zu jedem Zeitpunkt maximal der vom Benutzer angegebenen Anzahl der gleichzeitig darstellbaren Ausgabefenster. Die möglicherweise notwendige Terminierung eines Ausgabefensters erfolgt durch Signalübertragung.

7.5 Ausnahmebehandlung

Dieses Kapitel beschäftigt sich näher mit der Implementierung der Mechanismen zur Ausnahmebehandlung (siehe [DFG^{+}92]). In den Kapiteln 7.2.1 und 7.2.2 wurde bereits kurz erwähnt, wie verschiedene Fehlerzustände in den Ein- und Ausgabeprozessen bis hin zu den Komponenten der ProSet-Bibliothek propagiert werden und dort zur Erzeugung einer ProSet-Ausnahme führen. Eingabeprozesse propagieren Fehlerzustände durch spezielle Nachrichten, die anstelle des eingegebenen Datums an den Elternprozeß (Eingabefunktion der C-Bibliothek) zurückgesendet werden. Die Eingabefunktion reagiert auf diese speziellen Nachrichten durch Rückgabe eines entsprechend ausgezeichneten Wertes über die C-Schnittstelle an die ProSet-Hülle, welches den ausgezeichneten Wert als Fehlermeldung interpretiert und eine entsprechende ProSet-Ausnahme erzeugt.

Sämtliche Fehlerzustände in den Eingabeprozessen werden auf diese Art und Weise in das ProSet-Programm propagiert, wobei folgende Fehlerzustände berücksichtigt werden.

- Der Benutzer drückt den `Abbruch`-Button.
- Der Benutzer schließt das Dialogfenster über die Menüleiste.
- Die Umgebungsvariable `DISPLAY` ist nicht gesetzt.
- Das Modulefile `prosetgui` ist nicht geladen.

Laut den Anforderungen in Kapitel 3 sollte ebenfalls eine ProSet-Ausnahme erzeugt werden, falls ein Prozeß, der ein Eingabefenster kontrolliert, ein `kill`-Signal empfängt. Diese Anforderung ist nicht erfüllbar, da es nicht möglich ist, den Empfang eines solchen Signals zu behandeln (siehe [Tan90, Seite 23]). Dementsprechend kann auch keine Ausnahmebehandlung eingeleitet werden.

Die Propagierung der Fehlerzustände in Ausgabeprozessen erfolgt über die Signalübertragung von Unix (siehe dazu [Tan90]). In Ausgabeprozessen können insgesamt zwei Fehlerzustände auftreten:

- Die Umgebungsvariable `DISPLAY` ist nicht gesetzt.
- Das Modulefile `prosetgui` ist nicht geladen.

Die Ausgabefunktion der C-Bibliothek reagiert bei Empfang eines Signales mit der Rückgabe eines Statuswertes an die ProSet-Hülle, der den aufgetretenen Fehlerzustand beschreibt. Anhand dieses Statuswertes kann das ProSet-Programm eine entsprechende Ausnahmebehandlung einleiten.

Die endgültige Implementierung der Konzepte zur Ausnahmebehandlung weicht also von der in Kapitel 4.5 geplanten Art und Weise ab. Die sprachspezifischen Mechanismen zur Ausnahmebehandlungen werden nicht in die C-Bibliothek integriert, sondern sind Bestandteile der entsprechenden Programm-Hüllen, in denen der Aufruf der C-Bibliotheks-Funktion erfolgt. Dies hat den Vorteil, daß nur eine C-Bibliothek für alle Sprachen zur Verfügung gestellt werden muß. Zudem kann durch diese sprachenunabhängige Propagierung von Fehlerzuständen eine Fehlerbehandlung in Programmen erfolgen, die in einer anderen Programmiersprache programmiert sind und die Funktionen der C-Bibliothek nutzen. Die Art der Fehlerbehandlung hängt dann unter anderem von der gewählten Programmiersprache ab.

Folgendes Beispiel soll verdeutlichen, wie eine konkrete Ausnahmebehandlung in einem ProSet-Programm mit graphischer Ein- bzw. Ausgabe aussehen kann:

```
program Prog;
persistent constant PS_XGet,
                    PS_XPut:"StdLib.Xio";
begin

  a := PS_XGet() when Cancel use MyCancelHandler();
  PS_XPut(a)   when NoDisplay use MyDisplayHandler();

  handler MyCancelHandler();
  begin
    put("Cancel gedrueckt");
  end MyCancelHandler;

  handler MyDisplayHandler();
  begin
    put("Display nicht gesetzt");
  end MyDisplayHandler;

end Prog;
```

Das oben abgebildete ProSet-Programm erzeugt zunächst mit Hilfe von `PS_XGet()` einen Eingabe-Dialog, mit dessen Hilfe der Benutzer einen ProSet-Wert eingeben kann. Der eingegebene Wert wird in der Variablen `a` abgelegt. Anschließend wird der eingegebene Wert mit Hilfe der Anweisung `PS_XPut()` ausgegeben.

Drückt der Benutzer während der Eingabeaufforderung den `Abbruch`-Button des Eingabe-Dialoges, so springt das ProSet-Programm in den dafür angegebenen Handler `MyCancelHandler()`. Ähnliches passiert bei der darauf folgenden Anweisung zur Ausgabe eines ProSet-Wertes. Ist die Umgebungsvariable `DISPLAY` nicht gesetzt, so wird der dafür vorgesehene Handler `MyDisplayHandler()` aufgerufen.

Zur Behandlung der weiteren möglichen Ausnahmezustände (siehe dazu Kapitel 4.5) kann analog zu dem oben angegebenen Beispiel vorgegangen werden.

7.6 Tcl/Tk

Dieses Kapitel beschäftigt sich näher mit Tcl/Tk, insbesondere mit den Tcl-Skripten, die das Aussehen und das Verhalten der Ein- bzw. Ausgabe-Dialoge festlegen. Diese werden anhand eines einfachen Beispiels erläutert, ohne dabei auf programmiertechnische Details einzugehen. Um die folgenden Kapitel zu verstehen, sollte man die wesentlichen Sprachkonstrukte verstanden haben, die Tcl/Tk anbietet. Dazu verweisen wir auf [Ous94].

7.6.1 Tcl-Skripte zur Eingabe von ProSet-Werten

Wir betrachten im folgenden das Tcl-Skript zur Standard-Eingabe von ProSet-Werten.

```
set libpath $env(TCL_BIBS)
set auto_path [linsert $auto_path 0 $libpath]
```

Um die Funktionen unserer Tcl-Bibliotheken nutzen zu können, müssen wir den Pfad der entsprechenden Bibliotheken bekannt machen. Der Aufruf der Funktionen erfolgt nach dem Prinzip des *Autoloading* (siehe dazu [Ous94, Abschnitt 13.7]). Auf die einzelnen Bibliotheken gehen wir noch genauer in Kapitel 7.11 ein.

```
set x "Pinboard $PinID";
```

`x` bezeichnet den Namen des Prozesses, der das Pinboard realisiert. Dieser wird benötigt, um die Kommunikation zwischen Eingabeprozeß und Pinboard zu ermöglichen.

```
proc Insert Ins {
 .et delete 0 end
 set Trans [UnTransform $Ins];
 .et insert 0 $Trans
 if {[IsKorrekt [.et get]] == 1} {.btok config -state normal}\
 else {.btok config -state disabled}
}
```

`Insert` bezeichnet eine Funktion, die den gewünschten Copy-and-Paste-Mechanismus zwischen Pinboard und Eingabeprozeß realisiert. Wählt der Benutzer während einer Eingabeaufforderung ein Datum des Pinboards, anstatt ein neues Datum einzugeben, so wird die Funktion `Insert` des aktuellen Eingabe-Dialoges aufgerufen. Der Parameter `Ins` beinhaltet dabei das einzufügende Datum in transformierter Darstellung (siehe dazu 7.10). Sobald das Datum durch die Funktion `UnTransform` umgewandelt ist, wird es dialogabhängig in das zugehörige Dialogfenster eingetragen. In diesem Falle muß das Datum lediglich in ein dafür vorgesehenes Textfeld (`.et`) eingefügt werden. Das soeben eingefügte Datum wird dann hinsichtlich eines korrekten ProSet-Typen überprüft und der `Ok`-Button (`.btok`) entsprechend aktiviert bzw. deaktiviert.

Die Funktion `Insert` bildet einen fixen Bestandteil von Tcl-Skripten zur Eingabe von ProSet-Werten. Der Name dieser Funktion ist mit `Insert` fest vorgegeben, während der Inhalt von `Insert` je nach Dialog variieren kann.

```
scrollbar .scrollet -orient horizontal -command ".et xview"

entry .et -relief sunken -textvariable Inp \
                         -xscrollcommand ".scrollet set"
```

Mit Hilfe dieser beiden Anweisungen wird das benötigte Textfeld (`.et`) zusammen mit einem horizontalen Rollbalken (`.scrollet`) erzeugt. Die Textvariable `Inp` enthält zu jedem Zeitpunkt den Inhalt des Textfeldes `.et`.

```
button .btcancel -relief raised \
                 -text "Abbruch" \
                 -command \
                       {
                         if {[lsearch [winfo interps] $x] != -1} {
                                 send $x {set SetInp 0;fillall}
                         }
```

```
                        Sende 4;
                        exit
                      }
```

Diese Anweisung erzeugt einen Button mit der Beschriftung `Abbruch`. Wird dieser Button gedrückt, bricht der Benutzer die Eingabeaufforderung ab. Die Variable `SetInp` im Pinboard wird auf 0 gesetzt. Anhand dieser Variablenbelegung erkennt das Pinboard, daß keine Eingabeaufforderung vorliegt, und kann entsprechende Funktionen wie `Eingeben` deaktivieren. Durch die Anweisung `fillall` wird ein durch die Eingabeaufforderung aktivierter Filter wieder deaktiviert. Bevor die Eingabeaufforderung endgültig beendet wird, wird mittels der Funktion `Sende` eine Nachricht des Typs 4 an den Elternprozeß versendet, um dort eine Ausnahmebehandlung einzuleiten.

```
button .help -relief raised \
             -text "Hilfe" \
             -command {
                       Help "StringEingabe"
                      }
```

Ein Button mit der Aufschrift `Hilfe` wird durch diese Anweisung erzeugt. Auf Knopfdruck wird mit Hilfe des Kommandos `Help` (siehe Kapitel 7.3) das Hilfesystem gestartet.

```
if {$Show == 0} {
 button .btok -text "Ok"\
             -command \
                 {
                  set Input $Inp;
                  set Trans [Transform $Input];
                  if {[lsearch [winfo interps] $x] != -1} {
                         send $x finsert $Trans Standard-Eingabe;
                         send $x {set SetInp 0;fillall}
                  }
                  Sende 1
                  exit
                 }
}
```

Für den Fall, daß es sich bei dem erzeugten Dialog nicht um einen reproduzierten Dialog handelt (`Show`=0), wird ein `Ok`-Button spezifiziert, auf dessen Knopfdruck folgende Anweisungen durchführt werden:

- Belege die Variable `Input` mit dem aktuellen Inhalt des Textfeldes (siehe dazu Kapitel 7.2.1).
- Transformiere das eingegebene Datum (siehe dazu Kapitel 7.10).
- Füge das Datum zusammen mit dem Namen des Formulars und eventuellen weiteren Darstellungsattributen in das Pinboard ein. Dies geschieht mit der Funktion `finsert`.

- Setze die Variable `SetInp` auf 0 und deaktiviere den Datenfilter im Pinboard.
- Sende das eingegebene Datum an den Elternprozeß (Nachrichtentyp 1).
- Beende den Dialog.

```
if {$Show == 1} {
 button .btok -text "Ende" -command "exit"
}
```

Handelt es sich jedoch um einen reproduzierten Dialog (`Show`=1), so wird anstelle des `Ok`-Buttons ein `Ende`-Button erzeugt, der auf Knopfdruck lediglich den Dialog beendet, da bei reproduzierten Dialogen eine Eingabe nicht sinnvoll ist.

```
if {$Show == 0} {
               .btok config -state disabled
               bind .et <Any-KeyPress> \
                           {
                            if {[IsKorrekt [.et get]] == 1} {
                                     .btok config -state normal
                            } else {.btok config -state disabled}
                           }
}
```

Bei nicht reproduzierten Dialogfenstern soll bei jeder Änderung der Eingabe durch den Benutzer überprüft werden, ob das bis dahin eingegebene Datum ein korrektes ProSet-Datum ist (siehe dazu Kapitel 7.9). Abhängig davon wird der `Ok`-Button aktiviert bzw. deaktiviert.

```
pack .et -padx 2m -pady 1m -expand 1 -fill x
pack .scrollet -padx 2m -expand 1 -fill x
pack .btok -side bottom -side left -padx 2m \
           -pady 1m -expand 1 -fill x
pack .btcancel -side bottom -side left -padx 2m \
               -pady 1m -expand 1 -fill x
pack .help -side bottom -side left -padx 2m \
             -pady 1m -expand 1 -fill x
```

Diese Anweisungen dienen lediglich zur Festlegung der Geometrie des Dialoges und sollen hier nicht näher erläutert werden. Wir verweisen hierfür auf [Ous94].

```
if {$Show == 0} {
 if {[lsearch [winfo interps] $x] != -1} {
              send $x {set SetInp 1;fillall}
 }
}
```

Zu Beginn einer Eingabeaufforderung wird die Variable `SetInp` auf 1 gesetzt. Das Pinboard stellt daraufhin die Copy-and-Paste-Funktion, welche durch die Funktion `Eingeben` realisiert wird, zur Verfügung. Zusätzlich wird im Pinboard ein dialogabhängiger Filter (`fillall`) aktiviert, so daß der Benutzer nur diejenigen Daten des Pinboards selektieren kann, die für eine Eingabe dieses Dialoges in Frage kommen.

```
if {$Show == 1} {
 wm title . "Alte Eingabe $PinID"
 .et insert 0 $Input
 .et config -state disabled
 .help config -state disabled
 .btcancel config -state disabled
}
```

Bei reproduzierten Dialogen wird lediglich die Titelleiste des Fensters geändert und sämtliche Dialogobjekte des Dialoges bis auf den `Ende`-Button deaktiviert.

Bei dem oben beschriebenen Dialog zur Eingabe eines ProSet-Wertes handelt es sich um einen sehr einfachen Dialog (Standard-Eingabe). Das zugehörige Tcl-Skript besteht demnach aus relativ vielen fixen Bestandteilen, die sich in anderen Tcl-Skripten zur Eingabe von ProSet-Werten wiederfinden lassen. Um die Menge der bestehenden Dialoge zu erweitern, empfehlen wir, dieses Tcl-Skript zu übernehmen und entsprechende Änderungen durchzuführen. Notwendige Änderungen betreffen im wesentlichen

- die Dialogobjekte, die benötigt werden, um einen ProSet-Wert einzugeben,
- die Marke des Hilfetextes, der bei Betätigen des `Hilfe`-Buttons durch das Hilfesystem dargestellt wird,
- die Prozedur `Insert`,
- den Aufruf der Funktion `finsert`,
- und den dialogabhängigen Filter.

Zudem verweisen wir auf Anhang B.

7.6.2 Tcl-Skripte zur Ausgabe von ProSet-Werten

Im folgenden betrachten wir nun das Tcl-Skript zur Standard-Ausgabe von ProSet-Werten.

```
set libpath $env(TCL_BIBS)
set auto_path [linsert $auto_path 0 $libpath]
set OnlyOnce 0;
set x "Pinboard $PinID"
```

Diese Anweisungen entsprechen im wesentlichen den Anweisungen aus Kapitel 7.6.1. Zudem benutzen wir eine Variable `OnlyOnce`, deren Funktion wir später erklären werden.

```
proc Sende {} {
 global x;
 global Output;
 set Trans [Transform $Output];
 if {[lsearch [winfo interps] $x] != -1} {
  send $x finsert $Trans Standard-Ausgabe;
 }
 SetzeFertig;
}
```

Diese Funktion dient dazu, den ausgegebenen Wert zusammen mit dem Namen des Formulars und eventuellen Darstellungsattributen an das Pinboard zu senden. Mittels der Funktion `SetzeFertig` wird ein Signal an den Vaterprozeß gesendet, dessen Signal-Handler die globale Variable `OK` der C-Bibliothek auf `1` setzt (siehe dazu Kapitel 7.1.2).

```
scrollbar .scrollet -orient horizontal \
                    -command ".label xview"

entry .label  -relief ridge \
              -textvariable Output \
              -xscrollcommand ".scrollet set"
```

Zur Ausgabe eines ProSet-Wertes wird hier ein Textfeld mit einem horizontalem Rollbalken erzeugt. Aufgrund des Variablen-Links zwischen der Tcl-Variablen `Output` und der zugehörigen C-Variable gleichen Namens innerhalb des Ausgabeprozesses `Output`, enthält das Textfeld bereits zu Beginn den in `Output` enthaltenen Wert (das Ausgabedatum).

```
if {$Show == 0} {
 button .btquit -text "Ende" \
                -command {PidEntfernen;destroy .}
}
```

Für den Fall, daß es sich bei dem erzeugten Dialog nicht um einen reproduzierten Dialog handelt (`Show`=0), wird ein `Ende`-Button erzeugt, der auf Knopfdruck das erzeugte Dialogfenster zerstört. Vorher wird durch die Anweisung `PidEntfernen` der Prozeßidentifikator des Ausgabeprozesses, der das Dialogfenster kontrolliert, aus der Liste der Prozeßidentifikatoren im Pinboard entfernt (siehe dazu Kapitel 7.4).

```
if {$Show==1} {
 button .btquit -text "Ende" \
                -command "exit"
}
```

Handelt es sich jedoch um einen reproduzierten Dialog (`Show`=1), so wird anstelle des `Ende`-Buttons ein `Ende`-Button erzeugt, der auf Knopfdruck lediglich den Dialog beendet.

```
if {$Show == 0} {
 bind . <Expose> {
```

```
                  if {$OnlyOnce == 0} {
                   PidEinfuegen;
                   set OnlyOnce 1;
                   Sende
                  }
                 }
}
```

Sobald das Dialogfenster zur Ausgabe eines ProSet-Wertes auf dem Bildschirm dargestellt wird, wird der Prozeßidentifikator des Ausgabeprozesses in die Identifikatorenliste des Pinboards eingefügt (siehe dazu Kapitel 7.4). Da dies nur einmal geschehen soll, wird `OnlyOnce` auf 1 gesetzt. Ein einziges mal soll auch nur das ausgegebene Datum an das Pinboard gesendet werden.

```
if {$Show == 1} {
 wm title . "Alte Ausgabe $PinID"
}
```

Falls es sich bei dem Dialogfenster um ein reproduziertes Dialogfenster handelt, soll es die Titelleiste entsprechend anzeigen.

```
pack .et -padx 2m -pady 1m -expand 1 -fill x
pack .scrollet -padx 2m -expand 1 -fill x
pack .btquit -side bottom -side left -padx 2m -pady 1m \
             -expand 1 -fill x

bind .btquit <Destroy> {PidEntfernen}

.et config -state disabled
```

Hier wird die Geometrie der Dialogobjekte festgelegt. Bei der Zerstörung des Dialogfensters soll der Prozeßidentifikator aus der Liste der Prozeßidentifikatoren entfernt werden (`PidEntfernen`). Zudem wird das Textfeld deaktiviert, damit der Benutzer keine Eintragungen in dem Textfeld vornehmen kann.

Wie bei dem Tcl-Skript zur Eingabe eines ProSet-Wertes handelt es sich hier um einen sehr einfachen Dialog mit sehr vielen fixen Bestandteilen. Er kann für Erweiterungen mit entsprechenden Veränderungen übernommen werden.

7.7 Der Terminator

Dieses Kapitel beschäftigt sich mit der Realisierung des Terminators. In Kapitel 4.4 wurde bereits erwähnt, daß der Terminator als unabhängiger Unix-Prozeß realisiert werden sollte. Dieser Prozeß sollte im wesentlichen folgende Aufgaben erfüllen:

1. Schließen sämtlicher aktuellen und reproduzierten Dialogfenster

2. Löschen der IPC-Verwaltung

Wie weitere Überlegungen ergeben haben, liegt es nahe, diese Funktionen in das Pinboard zu integrieren und ganz auf den Terminator zu verzichten. Da das Pinboard bereits eine Liste von Prozeßidentifikatoren zur Überwachung der maximalen Anzahl gleichzeitig darstellbarer Ausgabefenster verwaltet, fällt es leicht, die erste der oben genannten Funktionen in das Pinboard zu integrieren. Wir erweitern das Pinboard um entsprechende Menüfunktionen, die in Kapitel 7.8 näher erläutert werden sollen.

Ein kompletter Verzicht auf den Terminator ist nicht möglich, da die IPC-Verwaltung Bestandteil der C-Bibliothek (und damit des ablaufenden ProSet-Programmes) ist und somit für das Pinboard als unabhängiger Unix-Prozeß schwer erreichbar ist.

Aufgrund der nur noch geringen Funktionalität des Terminators verzichten wir darauf, den Terminator als unabhängigen Unix-Prozeß zu realisieren. Damit verzichten wir auch auf das zugehörige Fenster, welches beim Aufruf des Terminators auf dem Bildschirm erscheinen sollte.

Der Terminator besteht endgültig lediglich als Komponente der C-Bibliothek, die zu jeder Zeit aufgerufen werden kann, mit dessen Aufruf jedoch nur die IPC-Verwaltung gelöscht wird.

Der Aufruf des Terminators innerhalb eines ProSet-Programmes geschieht mit der Anweisung `PS_Terminate()`. Wie bereits erwähnt, sollte der Aufruf am Ende eines ProSet-Programmes erfolgen.

7.8 Das Pinboard

In diesem Kapitel sollen wesentliche Grundlagen zur Arbeitsweise und Funktionalität des Pinboards erläutert werden. Zuvor soll jedoch das in Abbildung 7.4 dargestellte Aussehen des Pinboards diskutiert werden. Zur Informationsdarstellung der aufgezeichneten Daten werden drei Listboxen verwendet. Die Listbox unter dem Label `Datum` enthält die eigentlichen aufgezeichneten Daten, die während eines Programmablaufes ein- bzw. ausgegeben werden. Zwei weitere Listboxen stellen zu dem jeweiligen Datum das zugehörige Formular und eventuelle Darstellungsattribute dar (Überschriften: `Formular` und `Attribute`). Sämtliche Listboxen sind mit einem horizontalen Scrollbalken ausgestattet. Ein einziger vertikaler Scrollbalken dient dazu, den sichtbaren Bereich in allen drei Listboxen gleichzeitig zu verändern. Neben diesen Dialogobjekten stellt das Pinboard ein Menü mit Funktionen zur Verfügung, die in den folgenden Abschnitten näher erläutert werden.

7.8.1 Bearbeiten von Daten des Pinboards (Menüpunkt: `Bearbeiten`)

Unter diesem Menüpunkt sind folgende Funktionen auswählbar (siehe Abbildung 7.5):

- `Eingeben`
- `Anzeigen`
- `Loeschen`
- `Entfernen`
- `Alles waehlen`

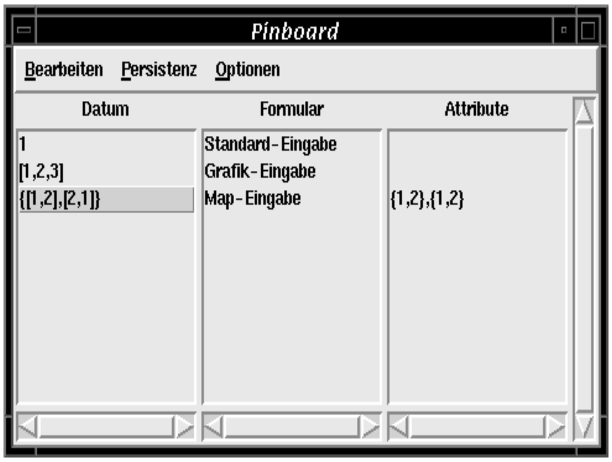


Abbildung 7.4: Das Pinboard

- `Beenden`

Wie in Kapitel 4.3 bereits beschrieben, verfügt das Pinboard über Funktionen zum Eingeben (`Eingeben`), Anzeigen (`Anzeigen`) und Entfernen (`Entfernen`) von selektierten Daten. Diese Funktionen sollen hier nicht noch einmal beschrieben werden. In Kapitel 7.7 wurde bereits erwähnt, daß eine Funktion des Terminators (Schließen sämtlicher aktuellen und reproduzierten Dialogfenster) in das Pinboard integriert wurde. Mit Hilfe des Menüpunktes `Löschen` kann diese Funktion durchgeführt werden. Dabei werden neben den dargestellten Dialogfenstern auch sämtliche Daten des Pinboards entfernt. Der Menüpunkt `Beenden` terminiert das Pinboard und schließt zudem sämtliche aktuellen und reproduzierten Dialogfenster.

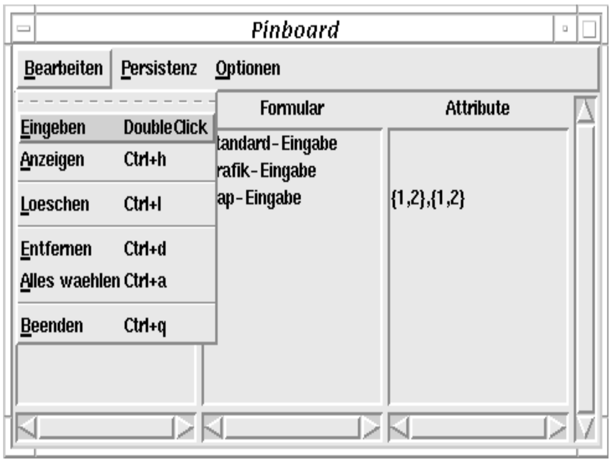


Abbildung 7.5: Bearbeiten von Daten des Pinboards

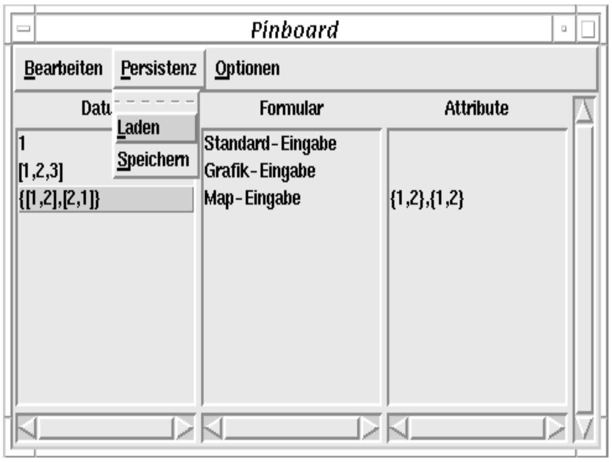


Abbildung 7.6: Persistenzfunktionen des Pinboards

7.8.2 Persistenzfunktionen des Pinboards (Menüpunkt: `Persistenz`)

Unter diesem Menp̈unkt sind folgende Funktionen auswählbar (siehe Abbildung 7.6):

- `Laden`
- `Speichern`

Das Pinboard stellt zwei Funktionen zur Verfügung, mit denen der Benutzer Daten eines P-Files laden kann (`Laden`) und die von ihm selektierten Daten in ein P-File ablegen kann (`Speichern`). Dabei werden die selektierten Daten als Tupel, also als ein einziges ProSet-Datum, abgespeichert bzw. geladen. Nachdem der Benutzer eine dieser Funktion gewählt hat, wird er aufgefordert, einen Bezeichner anzugeben. Für diese Zwecke erscheint das in Abbildung 7.8 dargestellte Fenster (Dieses Dialogfenster erscheint, nachdem der Benutzer den Menüpunkt `Laden` ausgewählt hat). Bestätigt der Benutzer die Eingabe des Bezeichners durch Drücken des `Ok`-Buttons, so werden die Elemente des unter diesem Bezeichner in dem Pfile `XioDaten` abgespeicherten Tupels in das Pinboard zusätzlich aufgenommen und stehen dem Benutzer für den weiteren Programmablauf zur Verfügung. Die zu speichernden Elemente werden als Tupel abgespeichert, da bei einem erneuten Laden die Reihenfolge (zeitliche Ein- und Ausgabefolge) dieser Daten beibehalten werden soll. Eine Ordnung von Elementen ist jedoch laut mathematischer Definition nur bei Tupeln und nicht bei Mengen gegeben.

Mit Hilfe der Funktion `Speichern` kann der Benutzer die von ihm selektierten Daten unter einem von ihm gewählten Bezeichner in dem P-File `XioDaten` ablegen. Dabei erscheint wiederum ein Dialogfenster, mit dessen Hilfe der Benutzer den von ihm gewünschten Bezeichner angeben kann.

Das Laden und Speichern von Daten erfolgt nicht im Pinboard selbst, sondern mittels einem dafür vorgesehenen P-File-Editor `xpfed`, der aufbauend auf Ideen in [Kap95] entwickelt wurde. Die dafür notwendigen Kommunikationsmechanismen werden durch `Tool-Talk` (siehe dazu [Sun93]) zur Verfügung gestellt. Dabei handelt es sich um ein Werkzeug, welches die Kommunikation zwischen mehreren Unix-Prozessen ermöglicht. Die Anbindung der Tool-Talk-Funktionalitäten an das Pinboard und den P-File-Editor erfolgt über eine entsprechende

Tcl-Bibliothek (`Tcl-Tool-Talk`). Abb. 7.7 beschreibt die Kommunikation zwischen dem Pinboard und dem P-File-Editor.

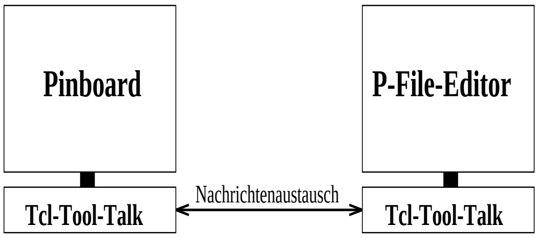


Abbildung 7.7: Kommunikation zwischen dem Pinboard und dem P-File-Editor mittels `Tool-Talk`

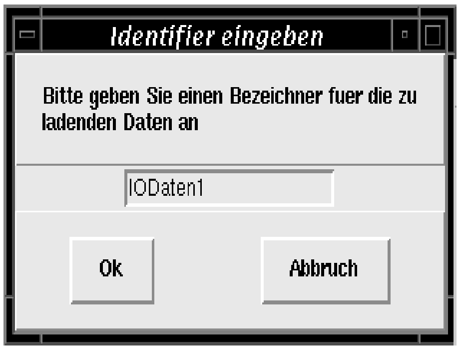


Abbildung 7.8: Eingabe eines Identifiers

7.8.3 Andere Funktionen des Pinboards (Menüpunkt: `Optionen`):

Unter diesem Menüpunkt ist folgende Funktion auswählbar (siehe Abbildung 7.9):

- `Fensteranzahl`

Zur Eingabe der maximalen Anzahl gleichzeitig darzustellender Ausgabe-Dialoge, wird der Menüpunkt `Fensteranzahl` angeboten. Nach Auswahl dieses Menüpunktes erscheint der in Abbildung 7.10 dargestellte Dialog, mit dessen Hilfe der Benutzer eine positive, ganze Zahl eingeben kann. Diese Zahl entspricht dann für den weiteren Programmablauf der Anzahl der maximal gleichzeitig darstellbaren Ausgabefenster.

7.9 Prüfung von Eingabedaten

Laut den ProSet-spezifischen Anforderungen in Kapitel 3 dürfen bei einer Eingabeaufforderung nur korrekte ProSet-Werte als Eingabe akzeptiert werden. Der `Ok`-Button eines

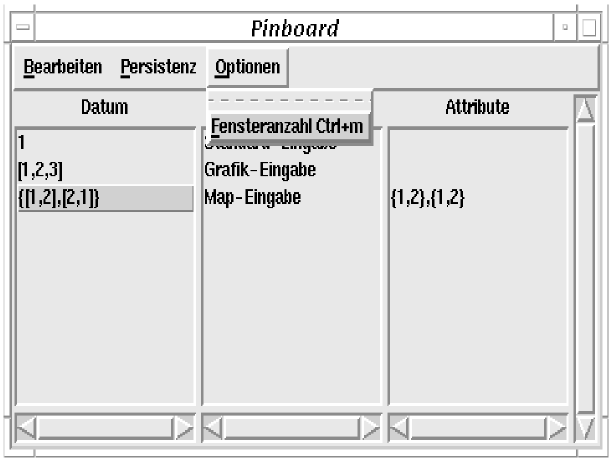


Abbildung 7.9: Das Menü `Optionen` des Pinboards

Abbildung 7.10: Eingabe einer maximalen Fensteranzahl

Eingabe-Dialoges wird dementsprechend aktiviert bzw. deaktiviert. Die Prüfung eines Eingabedatums wird bei jeder Modifikation des aktuellen Eingabedatums durchgeführt.

Die Prüfung des aktuellen Eingabedatums erfolgt mit Hilfe *regulärer Ausdrücke* (siehe dazu [Ous94]), sofern es sich bei dem zu prüfenden Eingabedatum um ein ProSet-Datum primitiven Typs handelt (siehe Kapitel 1.5). Bei komplexen ProSet-Daten (Mengen und Tupel) kann eine Prüfung auf der Basis regulärer Ausdrücke nicht stattfinden, da es sich hier um *Wörter* einer Sprache handelt, die nicht regulär ist, und die somit nicht durch reguläre Ausdrücke beschreibbar sind. Die sogenannte Dyck-Sprache

$$L = a^n b^n$$

beschreibt beliebige Klammerausdrücke, somit auch komplexe ProSet-Daten (Mengen und Tupel). Diese Sprache ist laut Pumping-Lemma nicht regulär (siehe [HU93]). Deren Wörter sind demnach nicht durch einen regulären Ausdruck beschreibbar.

Mit Hilfe der folgenden Feststellung scheint eine Prüfung von komplexen ProSet-Daten dennoch möglich:

Ein komplexes ProSet-Datum ist genau dann korrekt, wenn:

1. es von gleichartigen (eckigen und geschweiften) Klammern eingegrenzt ist,
2. seine Elemente durch Kommata getrennt sind und
3. jedes Element des Datums wiederum ein korrektes ProSet-Datum ist.

Ersteres kann mit Hilfe von regulären Ausdrücken geprüft werden. Anschließend werden sämtliche Elemente des zu prüfenden Datums einzeln auf ihre Korrektheit hin überprüft. Da es sich bei diesen Elementen wiederum um komplexe ProSet-Daten handeln kann, wurde ein rekursiver Algorithmus entworfen und implementiert, der als folgende Tcl-Funktion zur Verfügung steht:

`IsKorrekt` < `arg` >

Diese Funktion liefert den Wert 1 zurück, falls es sich bei der Zeichenkette `<arg>` um eine korrekte Darstellung eines ProSet-Datums handelt. Ansonsten liefert diese Funktion den Wert 0 zurück.

7.10 Aufgetretene Probleme

Transformation komplexer Daten: Komplexe ProSet-Daten (Mengen und Tupel) werden durch Mengenklammern bzw. durch eckige Klammern eingegrenzt. Dies führt zu Problemen, da diese Art von Klammern in Tcl ihre eigene Bedeutung besitzt. Insbesondere in Verbindung mit dem Tcl-Kommando `send` kam es zu ungewollten Interpretationen von Klammern. Folgendes Beispiel soll diesen Sachverhalt verdeutlichen:

```
send app puts $Datum
```

Diese Tcl-Anweisung sorgt dafür, daß eine Applikation namens `app` den Inhalt der Variablen `Datum` ausgibt. Enthält die Variable `Datum` beispielsweise das Datum `{1,2,3}` so führt die Applikation `app` folgende Anweisung durch:

```
puts {1,2,3}
```

Tcl interpretiert die Mengenklammern jedoch nicht als Bestandteil des Datums, sondern als überflüssige Begrenzer des Datums `1,2,3`. Dies hat schließlich die Ausgabe von `1,2,3` und nicht von `{1,2,3}` zur Folge. Zur Lösung dieses Problems wurden zwei Funktionen `Transform` und `UnTransform` implementiert. Mit Hilfe der Funktion `Transform` kann das Datum zunächst in eine `send`-verträgliche Form transformiert werden. Anschließend kann die Zielapplikation dieses transformierte Datum mit Hilfe der Funktion `UnTransform` in seine ursprüngliche Darstellung zurücktransformieren. Immer dann, wenn es zu Problemen mit der Doppeldeutigkeit von Klammern kommen kann, sollte der Benutzer Gebrauch von diesen Funktionen machen.

Tk 3.6 vs. Tk 4.0: Aufgrund von Inkompatibilitäten zwischen den Toolkit-Versionen Tk 3.6 und Tk 4.0 mußten während der Implementierungsphase diverse Änderungen innerhalb der Tcl-Skripten durchgeführt werden. Laut Ousterhout soll jedoch mit Tk 4.0 ein gewisser Standard erreicht worden sein, so daß neuere Versionen des Toolkits Tk keine weiteren größeren Änderungen in den Tcl-Skripten mit sich bringen dürften.

7.11 Tcl-Bibliotheken

Im folgenden sollen kurz die konstruierten Tcl-Bibliotheken, deren Komponenten von den Tcl-Skripten zur Ein- und Ausgabe von ProSet-Werten genutzt werden, beschrieben werden. Zur Erweiterung der Dialoge muß gegebenenfalls auf diese Komponenten zurückgegriffen werden.

Die Tcl-Bibliothek `libio.tcl`

Die Tcl-Bibliothek `libio.tcl` enthält folgende Komponenten:

Parse <Datum> Die Funktion `Parse` liefert für ein ProSet-Datum komplexen Typs (Menge oder Tupel) eine Tcl-Liste sämtlicher Elemente des Datums zurück. Wird beispielsweise `Parse` mit dem Datum `{1,{2,3,4}}` als Parameter aufgerufen, so liefert diese Funktion eine Tcl-Liste mit den Elementen `1` und `{2,3,4}` zurück.

Transform <Datum> Diese Funktion liefert das übergebene Datum in transformierter Darstellung zurück (siehe dazu Kapitel 7.10).

UnTransform <Datum> Diese Funktion liefert für ein Datum in transformierter Darstellung das entsprechende Datum in nicht-transformierter Darstellung zurück (siehe dazu ebenfalls Kapitel 7.10).

IsKorrekt <Datum> Diese Funktion wurde bereits in Kapitel 7.9 beschrieben.

PidEinfuegen Nach Aufruf dieser Funktion wird der Prozeßidentifikator des aufrufenden Prozesses, der das zugehörige Dialogfenster kontrolliert, in die Liste der Prozeßidentifikatoren aufgenommen (siehe dazu Kapitel 7.4).

PidEntfernen Diese Funktion entfernt den Prozeßidentifikator aus der Liste der Prozeßidentifikatoren, die vom Pinboard verwaltet wird (siehe dazu ebenfalls Kapitel 7.4).

Die Tcl-Bibliothek `user.tcl`

Diese Tcl-Bibliothek enthält folgende Komponenten:

show Innerhalb dieser Prozedur müssen mit der Erweiterung der Dialoge die Reproduktionsmechanismen erweitert werden. Nähere Informationen hierzu enthält Kapitel 7.12, sowie Anhang B.

Laden Diese Funktion dient zum Laden von persistenten Daten. Sie wird vom Pinboard aufgerufen, muß aber bei Erweiterungen durch den Benutzer ebenfalls erweitert werden (siehe Kapitel 7.12, sowie Anhang B).

Die Tcl-Bibliothek `help.tcl`

Folgende Funktion wird durch diese Bibliothek zur Verfügung gestellt:

Help Diese Funktion wurde bereits in Kapitel 7.3 beschrieben.

Die Tcl-Bibliothek `filter.tcl`

Diese Bibliothek enthält sämtliche Filter, die vom Pinboard zur Filterung der aufgezeichneten Daten benötigt werden. Je nach Erweiterung der Dialoge muß diese Bibliothek vom Benutzer entsprechend erweitert werden (siehe Kapitel 7.12, sowie Anhang B).

7.12 Die Konstruktion graphischer Dialoge

In diesem Kapitel sollen die wesentlichen Schritte zur Erweiterung der Bibliothek graphischer Dialoge noch einmal beschrieben werden.

Erweiterung der C-Bibliothek: Die C-Bibliothek `Xio.c` muß zunächst um die entsprechende Komponente erweitert werden. Die Komponenten der C-Bibliothek sollten das in Kapitel 7.1 beschriebene Aussehen besitzen. Gleichzeitig muß das Headerfile `Xio.h` um den entsprechenden Prozedur-Kopf ergänzt werden. Beispiele für die Komponenten der C-Bibliothek befinden sich in Kapitel 7.1, sowie in Anhang B.

Erzeugung eines Tcl-Skriptes: Als nächstes sollte das zum Dialog gehörige Tcl-Skript implementiert werden. Dazu empfehlen wir, das in Kapitel 7.6.1 bzw. 7.6.2 beschriebene Skript zu übernehmen und entsprechend anzupassen. Ein Beispiel hierfür befindet sich in Anhang B

Anpassung des Pinboards: Ein weiterer Schritt besteht in der Integration folgender Mechanismen in das Pinboard (Die letzten beiden Punkte müssen nur dann berücksichtigt werden, falls es sich bei dem neuen Dialog um einen Eingabe-Dialog handelt. Ein Beispiel zur Anpassung der unten aufgeführten Komponenten befindet sich ebenfalls in Anhang B.):

- Die Prozedur `show` innerhalb der Tcl-Bibliothek `user.tcl` muß derart erweitert werden, daß sich der neue Dialog mit Hilfe des Pinboards reproduzieren läßt. Für ein Beispiel sei wiederum auf Anhang B verwiesen.
- Damit das Pinboard die aufgezeichneten Daten dialogabhängig filtern kann, muß gegebenenfalls ein neuer Filter implementiert werden. Falls die Tcl-Bibliothek `filter.tcl` für den zu konstruierenden Dialog noch keinen Filter zur Verfügung stellt, muß sie entsprechend um einen neuen Filter ergänzt werden.
- Falls ein neuer Filter implementiert wird, muß die Tcl-Prozedur `Laden`, die sich ebenfalls in der Bibliothek `user.tcl` befindet, entsprechend angepaßt werden.

Erweiterung des Hilfesystems: Die Hilfedatei `PSXio.hlp` muß als nächstes um den für den zu konstruierenden Dialog vorgesehenen Hilfetext ergänzt werden (siehe Anhang B).

Erweiterung der ProSet-Bibliothek: Zum Schluß wird die ProSet-Bibliothek um eine Komponente ergänzt. Diese Komponente bildet eine ProSet-spezifische Hülle für den Aufruf der neu konstruierten Komponente der C-Bibliothek. Auch hier verweisen wir auf den Anhang B.

Kapitel 8

Formale Spezifikation graphischer Dialoge

Nachdem in Kapitel 7 eingehend auf die Implementierung graphischer Dialoge eingegangen worden ist, soll in diesem Kapitel der formale Aspekt bei der Entwicklung graphischer Benutzungsschnittstellen betrachtet werden (formale Spezifikation). Die hier verwendete Spezifikationsmethode soll zunächst in Kapitel 8.1 beschrieben und an einem Beispiel verdeutlicht werden. Anhand eines Beispieldialoges wird dann in Kapitel 8.2 konkret gezeigt, wie das Ergebnis der Umsetzung eines formal spezifizierten Dialoges aussehen kann. Zudem erfolgt in Kapitel 8.3 eine Bewertung der vorgestellten Spezifikationsmethode.

8.1 Eine Spezifikationsmethode

Eine *formale Spezifikation* dient in erster Linie dazu, Syntax und Semantik von Software-Komponenten (bspw. abstrakter Datentypen oder graphischer Benutzungsschnittstellen) formal zu beschreiben. Eine Spezifikationsmethode sollte einerseits möglichst formal sein, um eine weitgehende Beweisbarkeit auf der Spezifikation beruhender Implementierungen zu ermöglichen, andererseits jedoch soll sie auch verständlich sein, um dem Ziel gerecht zu werden, eine verständliche Spezifikation der Software-Komponente zu konstruieren. Der Programmierer, der formal spezifizierte Software-Komponenten realisiert, kann in der Regel selbst entscheiden, wie er die entsprechenden Komponenten implementiert, solange er die vorgegebene Spezifikation nicht verletzt.

Formale Spezifikationskonzepte orientieren sich in der Regel an einem *Modell.* Ein Beispiel sind *algebraische Spezifikationsmethoden*, mit deren Hilfe formale Definitionen von abstrakten Datentypen mit ihren Operationen und ihrer Semantik möglich sind. Als Modell dienen hier *mehrsortige Algebren* (siehe bspw. [See90]).

Im folgenden wird nun eine Spezifikationssprache vorgestellt, mit deren Hilfe es möglich ist, graphische Benutzungsschnittstellen formal zu spezifizieren. Ziel dieser formalen Spezifikation ist in erster Linie eine Beschreibung der graphischen Schnittstelle aus Sicht des Benutzers und weniger aus Sicht des Programmierers (siehe [Jac86]). Die Spezifikationsmethode basiert auf der Beobachtung, daß graphische Benutzungsschnittstellen eine *Koroutinen*-ähnliche Struktur und verschiedenste Dialogzustände besitzen. Jeder geeignete Teildialog wird als separates Objekt durch ein eigenes Zustandsdiagramm beschrieben, welches suspendiert und wieder aufgenommen werden kann und seinen Zustand beibehält. Die Kombination aller derartigen Objekte definiert die gesamte Schnittstelle als eine Menge von Koroutinen, wodurch eine Spezifikation als ein einziges komplexes Zustandsübergangsdiagramm im Sinne einer besseren Übersichtlichkeit vermieden wird. Ferner unterstützt die Spezifikationsmethode *Vererbung*, *Komponentenbildung* von Objekten sowie sogenannte *synthetische Token*. Die Beschreibung und Bedeutung dieser Konzepte erklären wir in den folgenden Kapiteln genauer.

8.1.1 Ein einführendes Beispiel

In diesem Kapitel soll die oben eingeführte Spezifikationsmethode zusammen mit einer Ada-basierten Notation anhand eines einfachen Beispiels erläutert werden (siehe dazu auch [Jac86]).

Es soll ein Button spezifiziert werden, der immer dann invertiert wird, wenn der Mauszeiger den zugehörigen Bildschirmausschnitt (Button-Fläche) betritt. Verläßt der Mauszeiger diesen Bereich, so nimmt der Button seinen ursprünglichen Zustand wieder ein. Wird die linke Maustaste gedrückt, während sich der Mauszeiger über der Button-Fläche befindet, so wird eine Routine namens `Return()` aufgerufen. Die Spezifikation des beschriebenen Interaktionsobjektes besteht aus einer Kombination eines Zustandsübergangsdiagrammes mit einer einfachen Ada-basierten Notation. Die Spezifikation hat folgendes Aussehen:

INTERACTION-OBJECT OkButton **is**

IVARS:

position	:= {100, 200, 64, 24}; - -*Koordinaten des Bildschirmausschnitts*
bgrc	:= Red; - -*Hintergrundfarbe*
sensitive	:= True; - -*Button ist sensitiv*

METHODS:

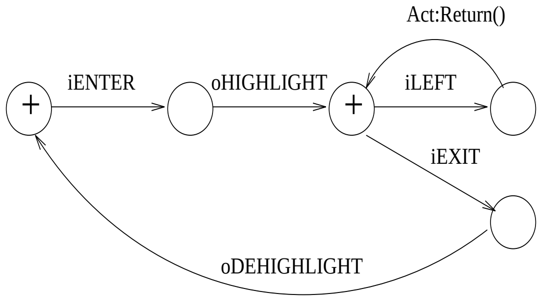
Draw()	{`DrawTextButton(position, "Ok");`}

TOKENS:

iLEFT	{ - -*Clicken der linken Maustaste*- - }
iENTER	{ - -*Cursor bewegt sich in das durch* `position` *gegebene Rechteck*- - }
iEXIT	{ - -*Cursor verläßt das durch* `position` *gegebene Rechteck*- - }
oHIGHLIGHT	{ - -*Invertieren des durch* `position` *gegebenen Rechtecks*- - }
oDEHIGHLIGHT	{ - -*Invertieren des durch* `position` *gegebenen Rechtecks*- - }

SYNTAX:

main



end INTERACTION-OBJECT;

Die Komponenten der Notation haben im einzelnen folgende Bedeutung:

IVARS: Eine Liste von Variablen zusammen mit ihren initialen Werten. Derartige Variablen können wie in diesem Beispiel die Position des Interaktionsobjektes sowie dessen Hintergrundfarbe festlegen. Sie können jedoch auch andere Interaktionsobjekte bezeichnen, die als Komponententeile dieses Objektes verwendet werden (siehe Kapitel 8.1.2).

METHODS: Für dieses Objekt lokal eindeutige Prozedur-Definitionen. In diesem Beispiel ist die Anweisung zum Zeichnen des Buttons abhängig von seiner Position und von seiner Beschriftung spezifiziert. Es können aber auch allgemeinere Prozeduren von geerbten Interaktionsobjekten (siehe Kapitel 8.1.2) spezialisiert werden. Auch können hier Prozeduren zur Erzeugung und Zerstörung des Interaktionsobjektes aufgeführt werden.

TOKENS: Der Name `Token` steht hier für Ein- bzw. Ausgabeereignisse einer graphischen Benutzungsschnittstelle. Hier wird jedes derartige Token, welches im Syntaxdiagramm des Interaktionsobjektes verwendet wird, definiert. Die Definition kann in natürlicher Sprache, einer konventionellen Programmier-Sprache oder mit Hilfe einer einfachen Notation angegeben werden. Input-Token werden hier mit dem Buchstaben `i`, Output-Token mit dem Buchstaben `o` gekennzeichnet.

SYNTAX: Der Eingabe-Handler für dieses Interaktionsobjekt, beschrieben durch ein Zustandsübergangsdiagramm, welcher als Koroutine zusammen mit den übrigen Spezifikationen ausgeführt wird. Jede Zustandstransition kann ein Input- oder Output-Token, den Namen eines anderen als Subroutine aufzurufenden Diagramms, eine auszuführende Aktion, eine zu testende Bedingung oder keine Beschriftung besitzen. Eine Transition, die keine Beschriftung besitzt, schaltet dann, falls keine andere Transition dieses Zustandsübergangsdiagrammes schalten kann. Eine Aktion, wie in diesem Beispiel `Return()`, ruft eine in der Applikation definierte Prozedur auf. In mit `+` markierten Zuständen kann der Dialog suspendiert und später wieder aufgenommen werden.

Weiterhin stehen dem Benutzer folgende Beschreibungskomponenten zur Verfügung:

FROM: Liste von Interaktionsobjekten, von denen dieses Interaktionsobjekt erbt (siehe Kapitel 8.1.2). Hierbei ist eine Ordnung gegeben: Elemente von früher aufgelisteten Objekten überschreiben die entsprechenden Elemente von später aufgelisteten.

SUBS: Zusätzliche Zustandsübergangsdiagramme, die als Unterprogramme vom Syntaxdiagramm aufgerufen werden können.

STATES: Liste von Transitionen, die in einem oder mehreren Zuständen des Zustandsübergangsdiagrammes ausgeführt werden können. Hiermit lassen sich Default-Übergänge definieren, die im Übergangsdiagramm aus Gründen der besseren Übersichtlichkeit nicht angegeben werden. Beispiel ist hier ein Hilfesystem, das in jedem Zustand durch Druck einer entsprechenden Taste aufgerufen werden kann.

8.1.2 Konzepte der Spezifikationsmethode

Vererbung: Beschreibungen von umfangreichen Benutzungsschnittstellen werden schnell komplex und Teile der Beschreibung wiederholen sich in regelmäßigen Abständen. Eine mögliche Lösung dieses Problems liegt in der Nutzung der *Vererbung*. Ein Interaktionsobjekt kann von anderen Interaktionsobjekten erben, indem es sie unter `FROM` auflistet. Dadurch fügt es zu seinen eigenen Definitionen alles hinzu, was unter den Punkten `IVARS`, `METHODS`, `TOKENS`, `SUBS` und `STATES` dieser Objekte steht, oder redefiniert es. Die Reihenfolge der Objekte in der `FROM`-Liste spielt eine wesentliche Rolle (siehe Kapitel 8.1.1). Vererbung kann auch über mehrere „Generationen“ geschehen und das Konzept der Mehrfachvererbung wird ebenfalls unterstützt, d.h. ein Interaktionsobjekt kann gleichzeitig von mehreren Interaktionsobjekten erben. Bei der Vererbung wird auch jeweils das gesamte Syntaxdiagramm vererbt.

Komponentenbildung: Interaktionsobjekte können als Kombination verschiedener anderer Interaktionsobjekte beschrieben werden. Im Gegensatz zur oben beschriebenen Vererbung *enthält* dieses Objekt dann die anderen Teilkomponenten, welche stets Interaktionsobjekte mit eigener Funktionalität sind. Erzeugung, Zeichnung und Zerstörung der enthaltenen Interaktionsobjekte erfolgt jeweils mit dem sie enthaltenden Objekt.

Synthetische Token: Mit Hilfe von synthetischen Token ist es möglich, Folgen von Ein- und Ausgabe-Operationen niedrigen Niveaus in größere Einheiten zusammenzufassen. Ein Objekt höheren Niveaus kann dann mit synthetischen Token arbeiten. In dem Beispiel aus Kapitel 8.1.1 existieren keine synthetischen Token. Vorstellbar wäre ein synthetisches Token, welches das Drücken und das Loslassen eines Mausknopfes als ein einziges Ereignis definiert. Dieses Token könnte dann bspw. von einem Interaktionsobjekt *Textfeld* verwendet werden.

8.2 Spezifikation und Implementierung eines Beispieldialoges

Nachdem in Kapitel 8.1 eine Methode zur formalen Spezifikation graphischer Benutzungsschnittstellen vorgestellt wurde, soll nun die Beschreibung einer Anwendung dieser Methode folgen. Ein wichtiges Kriterium einer späteren Bewertung soll die Implementierung eines formal spezifizierten Dialoges sein. Kapitel 8.2.1 enthält die formale Spezifikation des zu implementierenden Dialoges. Abschließend folgt in Kapitel 8.3 eine Bewertung der Methode zur formalen Spezifikation graphischer Benutzungsschnittstellen.

8.2.1 Spezifikation des Dialoges

Vorgegeben wird die folgende Spezifikation:

Formale Spezifikation für den *QuitButton* des Formulars:

INTERACTION-OBJECT QuitButton **is**

IVARS:

width	:= `concreteWidth`;	- -*Breite des Quit-Button*
height	:= `concreteHeight`;	- -*Höhe des Quit-Button*
position;		- -*Koordinaten des Bildschirmausschnitts*
legend	:= `"Quit"`;	- -*Beschriftung des Quit-Button*
bgrc	:= Red;	- -*Hintergrundfarbe*
fgc	:= DarkGreen;	- -*Vordergrundfarbe*

METHODS:

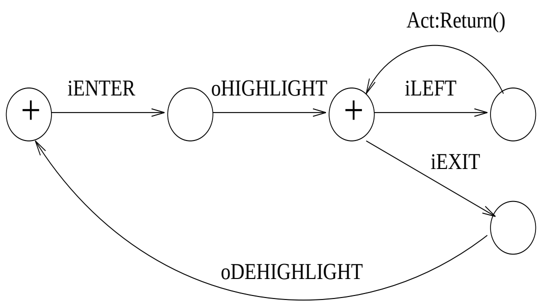
Draw()	{`DrawTextButton(position,width,height,legend);`}

TOKENS:

iLEFT	{ - -*Clicken der linken Maustaste*- - }
iENTER	{ - -*Cursor bewegt sich in das durch* `position` *gegebene Rechteck*- - }
iEXIT	{ - -*Cursor verläßt das durch* `position` *gegebene Rechteck*- - }
oHIGHLIGHT	{ - -*Invertieren des durch* `position` *gegebenen Rechtecks*- - }
oDEHIGHLIGHT	{ - -*Invertieren des durch* `position` *gegebenen Rechtecks*- - }

SYNTAX:

main



end INTERACTION-OBJECT;

Formale Spezifikation eines *generischen Textfeldes*:

INTERACTION-OBJECT GenericTextLabel **is**

IVARS:

bgc	:= LightBlue;	- -*Hintergrundfarbe des Textfeldes*
fgc	:= White;	- -*Schriftfarbe*

end INTERACTION-OBJECT;

Formale Spezifikation eines *Textfeldes* **des Formulars:**

INTERACTION-OBJECT TextLabel **is**

FROM:

GenericTextLabel;

IVARS:

width	:= `concreteWidth`;	- -*Breite des Textfeldes*
height	:= `concreteHeight`;	- -*Höhe des Textfeldes*
position;		- -*Koordinaten des Bildschirmausschnitts*
legend;		- -*Text*

METHODS:

Draw()	{ `DrawTextLabel(position,width,height,legend);` }

end INTERACTION-OBJECT;

Formale Spezifikation für das *Photograph* **des Formulars:**

INTERACTION-OBJECT Photograph **is**

IVARS:

width	:= `concreteWidth`;	- -*Breite des Fotos*
height	:= `concreteHeight`;	- -*Höhe des Fotos*
position;		- -*Koordinaten des Bildschirmausschnitts*
filename	:= `"Photograph.pic"`;	- -*Dateiname des Fotos*

METHODS:

Draw()	{ `DrawPicFile(position,width,height,filename);` }

end INTERACTION-OBJECT;

Formale Spezifikation des Formulars *PersonalFileForm*:

```
INTERACTION-OBJECT PersonalFileForm is

FROM:
 GenericMotifWindow; - -PersonalFileForm erbt Motif-Rahmen, Draw-Methode etc.
 Constants;      - -PersonalFileForm erbt Konstanten

IVARS:
 width      := concreteWidth;          - -Breite des Fensters
 height     := concreteHeight;         - -Höhe des Fensters
 position   := {concreteX1, concreteX2, concreteY1, concreteY2};
 bgc        := PeachPuff;              - -Fensterhintergrundfarbe
 surname    := new INTERACTION-OBJECT TextLabel
                                       (
                                        position =>
                                        position+[Xconst, Yconst];
                                        legend =>
                                        "Name: Puetter";
                                       );
 firstname  := new INTERACTION-OBJECT TextLabel
                                       (
                                        position =>
                                        position+
                                        [Xconst, 2*Yconst+Height(TextLabel)];
                                        legend =>
                                        "Vorname: Oliver";
                                       );
 occupation := new INTERACTION-OBJECT TextLabel
                                       (
                                        position =>
                                        position+
                                        [Xconst, 3*Yconst+2*Height(TextLabel)];
                                        legend =>
                                        "Beruf: Student";
                                       );
 hobby      := new INTERACTION-OBJECT TextLabel
                                       (
                                        position =>
                                        position+
                                        [Xconst, 4*Yconst+3*Height(TextLabel)];
                                        legend =>
                                        "Hobby: Motorradfahren";
                                       );
 picture    := new INTERACTION-OBJECT Photograph
                                       (
                                        position =>
                                        position+
                                        [2*Xconst+Width(TextLabel), Yconst];
```

```
                                   );
button     := new INTERACTION-OBJECT QuitButton
                                   (
                                    position =>
                                    position+
                                    [0.5*(concreteWidth-QuitButton.width)
                                    concreteHeight-QuitButton.height-Yconst];
                                   );


end INTERACTION-OBJECT;
```

Erläuterungen:

- Die Beschreibung des Interaktionsobjektes *PersonalFileForm* ist eine Kombination verschiedener Komponenten, die jeweils ein getrenntes Interaktionsobjekt beschreiben. Im Gegensatz zur Vererbung (siehe in diesem Fall das generische Interaktionsobjekt *GenericTextLabel*, von welchem das Interaktionsobjekt *TextLabel* **erbt**) **enthält** hier das Objekt *PersonalFileForm* u.a. mehrere Objekte des Typs *TextLabel*. Alle diese enthaltenen Objekte besitzen ihre eigene Beschreibung. Sie werden automatisch erzeugt, gezeichnet und zerstört wenn dies mit dem sie enthaltenden Objekt geschieht (die Komponentenobjekte können demnach also als Instanzvariablen des enthaltenden Objektes betrachtet werden).

- `Xconst` und `Yconst` dienen dazu, daß von den Komponentenobjekten zum Rand des Formulars etwas Abstand bleibt. Sie werden hier auch dazu verwendet, daß ein „Schönheitsabstand" zwischen den einzelnen Komponentenobjekten entsteht.

- `width` und `height` repräsentieren Breite und Höhe des Rechtecks, durch welches das entsprechende Interaktionsobjekt dargestellt wird.

- `position` stellt ein Tupel mit den Koordinaten der linken oberen Ecke und der rechten unteren Ecke des Rechtecks dar, durch welches das entsprechende Interaktionsobjekt dargestellt wird. Hat `position` keinen Wert, so wird dieser durch das das Komponentenobjekt enthaltende Objekt konkret festgelegt (siehe bspw. das in *PersonalFileForm* enthaltene Objekt *surname*).

Erweiterungen:

Beispielsweise könnte das Interaktionsobjekt *Photograph* zu einem *graphischen Button* erweitert werden, indem es von einem generischen Button *GenericButton* erbt und die darin noch unspezifizierte Interaktion durch ein Button-Click konkretisiert. Weiterhin könnten noch komplexere Interaktionsobjekte spezifiziert werden, bei denen dann auch über die Anbindung von ProSet-Variablen nachgedacht werden muß.

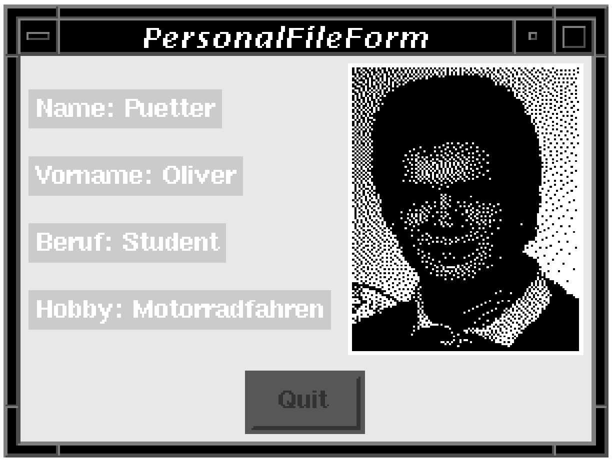


Abbildung 8.1: Dialogfenster des formal spezifizierten Dialoges

8.2.2 Implementierung mit Tcl/Tk

Die Implementierung des Dialoges erfolgte analog zu der in Kapitel 7.12 beschriebenen Vorgehensweise. Auf eine genaue Beschreibung der Implementierung soll hier verzichtet werden.

Ausgehend von der in Kapitel 8.2.1 angegebenene formalen Spezifikation entstand der in Abb. 8.1 dargestellte Dialog.

8.3 Erfahrungen und Bewertung der Spezifikationsmethode

Generell kann die in Kapitel 8.1 vorgestellte Methode zur formalen Spezifikation graphischer Benutzungsschnittstellen als angemessen erachtet werden. Formale Spezifikationen dieser Art sind für den Programmierer leicht verständlich und erlauben aufgrund der angebotenen Konzepte zur Modularisierung und Vererbung eine modulare und übersichtliche Beschreibung der zu realisierenden Schnittstelle. Weiterhin wurden folgende Erfahrungen gesammelt:

- Aufgrund der leichten Verständlichkeit der Spezifikation konnte das Aussehen und Verhalten des zu realisierenden Formulars schnell erkannt und realisiert werden.
- Defizite, wie fehlende Zwischenräume zwischen den einzelnen Komponentenobjekten, wurden schon in der Spezifikation erkennbar.
- Gerade in Verbindung mit Tcl/Tk kann die in Kapitel 8.1 vorgestellte Spezifikationsmethode aus folgenden Gründen als angemessen bewertet werden:
  - Viele der in der formalen Spezifikation angegebenen Instanzvariablen, wie z.B. Farben und Koordinaten der einzelnen Dialogobjekte, konnten in Tcl/Tk als Konfigurationsoptionen direkt angegeben werden, was die Implementierung der entsprechenden Objekte ausgehend von der formalen Spezifikation sehr vereinfachte.
  - Ähnliches gilt für die Implementierung des Verhaltens der einzelnen Widgets. Tcl/Tk bietet für viele der in der Spezifikation angegebenen Token (im Sinne von

[Jac86]) entsprechende Event-Typen an, so daß auch hier eine Implementierung ausgehend von der Spezifikation keine großen Schwierigkeiten bereitet.

Kapitel 9

Verwandte Arbeiten

Diese Arbeit beschreibt eine graphische Erweiterung der Prototyping-Sprache ProSet. Eine ähnliche Arbeit wurde im Rahmen des Software-Praktikums am Lehrstuhl 10 (Software-Technologie) der Universität Dortmund durchgeführt. Ziel dieser Arbeit war es, die Programmiersprache *ML* ([Wik87]) um eine graphische Benutzungsschnittstelle zu erweitern. Dabei sollte eine Bibliothek konstruiert werden, die dem *ML*-Programmierer eine möglichst einfache Schnittstelle zur Konstruktion graphischer Benutzungsschnittstellen zur Verfügung stellt. Die graphische Schnittstelle selbst wurde durch das Toolkit *Tk* realisiert. Die Erweiterung der Sprache *ML* um eine graphische Benutzungsschnittstelle geschah im Hinblick auf eine externe Kontrolle (ereignisgesteuerte Kontrolle) (siehe dazu [BG95]).

Im Rahmen einer Hiwi-Tätigkeit wurde am Lehrstuhl Praktische Informatik III der Fern-Universität Hagen eine Schnittstelle zwischen einer *Eiffel*-Implementierung ([Mey88]) und dem X-Window-System konstruiert. Mit dieser Schnittstelle war eine graphische *Ausgabe* von Daten basierend auf dem Konzept der internen Kontrolle beabsichtigt. Zur Realisierung wurde jedoch der ungenügende Ansatz 2 (mehrere Event-Schleifen) aus Kapitel 4.1 der vorliegenden Arbeit gewählt.

Desweiteren sind u.a. folgende Tcl- bzw. Tk-Erweiterungen von Programmiersprachen bekannt:

TkPerl5: *Perl5* (Practice Extraction and Report Language) ist eine nicht-kommerzielle objektorientierte Interpreter-Sprache, deren hauptsächliches Anwendungsgebiet in der Informations-Extraktion aus beliebigen Textdateien liegt (siehe dazu [WS93]). Ähnlich wie Tcl verfügte diese Sprache über keinerlei graphische Anbindung. Erst mit der Erweiterung *TkPerl5* wurde der Sprachumfang der Sprache derart erweitert, daß eine objektorientierte Programmierung graphischer Schnittstellen in Perl5 ermöglicht wurde.

ProTcl: *ProTcl* bezeichnet eine graphische Erweiterung der Programmiersprache *Prolog* ([CM81]). Im Gegensatz zur bereits beschriebenen ML-Erweiterung werden hier die Tcl- und Tk-Bibliotheken dynamisch an das *Prolog*-Programm gebunden, so daß es zusammen mit dem Tcl-Interpreter einen einzigen Prozeß bildet und so ein Kommunikations-Overhead zwischen dem *Prolog*-Programm und dem Tcl-Interpreter als eigenständiger Prozeß vermieden wird (Leichtgewicht-Interface).

Guile: *Guile* bezeichnet eine Erweiterung des Scheme-Interpreters SCM zu einem Multi-Sprachen-Interpreter. Dabei werden sämtliche von Guile unterstützten Sprachen mittels des Werkzeuges *yacc* in die entsprechende Scheme-Syntax überführt und dann interpretiert. Die zugehörige Unix-Bibliothek *libguile.a* ermöglicht die Verwendung des Interpreters *Guile* in gewöhnlichen C-Programmen. Bestandteil des Interpreters sind unter anderem sämtliche Tcl- und Tk-Funktionen, die als Build-In-Kommandos durch die C-Schnittstelle von Tcl/Tk zur Verfügung gestellt werden. So ist es beispielsweise möglich, innerhalb eines Scheme-Programmes einen Tcl-Interpreter zu erzeugen, um somit den Sprachumfang der Sprache Tcl zusätzlich zu nutzen. Mit Hilfe der Tk-Funktionen können zudem beliebige graphische Schnittstellen erzeugt werden. Andere Sprachen, die von Guile interpretiert werden können, besitzen dieselbe Möglichkeit. Für nähere Informationen verweisen wir auf [Lor95].

Modula-3: Die imperative sowie objektorientierte Programmiersprache *Modula-3* ([EK92]) nutzt ebenfalls sämtliche Build-In-Kommandos der Tcl- und Tk-Funktionen. Die Anbindung erfolgt über die C-Schnittstelle von *Modula-3*.

Nachdem im vorangegangenen Teil einige graphische Spracherweiterungen vorgestellt wurden, soll am Ende dieses Kapitels ein Datenbankmanagementsystem namens *Paradox* ([Age]) erwähnt werden, welches von Borland entwickelt wurde. Dieses Datenbankmanagementsystem verfügt bereits über eine Schnittstelle zu *Microsoft-Windows*, soll jedoch aufgrund einer Erweiterung namens *ezDialog* von Woll2Woll-Software hier Erwähnung finden. *ezDialog* bezeichnet ein Software-Paket, welches unter anderem eine Sammlung von graphischen Dialogen zur Integration in *Paradox*-Applikationen anbietet. Einige der mit diesem Paket angebotenen Dialoge wurden auch in dieser Arbeit realisiert, wobei ähnliche Entwurfsentscheidungen getroffen wurden.

Kapitel 10

Zusammenfassung und Ausblick

Mit dieser Diplomarbeit wurde eine graphische Ein- und Ausgabe von ProSet-Werten realisiert. Der Aufruf der Ein- und Ausgabeoperationen basiert auf dem Konzept der internen Kontrolle. Mit Hilfe eines Pinboards werden die in einem Programm ein- und ausgegebenen Daten aufgezeichnet und somit wiederverwendbar gemacht (interreferentielle Ein- und Ausgabe). Aufgezeichnete Daten können mit Hilfe des ProSet-Persistenzmechanismus persistent gemacht werden und stehen somit auch nach Beendigung eines Programmes weiterhin als Ein- und Ausgabedaten zur Verfügung. Ebenfalls wurden die ProSet-Mechanismen zur Ausnahmebehandlung integriert.

Die hier vorgestellten Ergebnisse bieten unter anderem folgende Erweiterungsmöglichkeiten:

Erweiterung der Dialoge: Die in dieser Diplomarbeit entworfene ProSet-Bibliothek zur Ein- und Ausgabe von ProSet-Werten kann um entsprechende Komponenten erweitert werden.

Transaktionsmechanismen: Das Pinboard kann um Transaktionsmechanismen erweitert werden. Mittels eines weiteren Menüpunktes könnten Transaktionen gestartet werden, so daß zu einem gewissen Zeitpunkt eine Folge von Ein- und Ausgaben durch Abbrechen der Transaktion rückgängig gemacht werden kann. Dabei werden dann die Daten, die nach dem Start der Transaktion ein- bzw. ausgegeben wurden, aus dem Pinboard entfernt.

Parallelität: Um die Parallelitätskonzepte von ProSet nutzen zu können, müssen ähnlich wie bei der textuellen Eingaberoutine `get` sämtliche graphischen Eingaberoutinen um einen Sperrmechanismus ergänzt werden, damit bei parallelen ProSet-Programmen zu jedem Zeitpunkt nur eine Eingabeaufforderung anliegen kann.

ProSet/Tk: Mit dieser Arbeit wurde die Prototyping-Sprache ProSet um eine graphische Benutzungsschnittstelle im Hinblick auf eine interne Kontrolle erweitert. Zudem konzentrierte sich die Erweiterung auf eine graphische Ein- und Ausgabe von ProSet-Werten. Eine komplette sowie ereignisorientierte Erweiterung von ProSet könnte, ähnlich wie bei Tcl, mit Hilfe des Toolkits Tk geschehen. Derartige Arbeiten sind in Vorbereitung.

Literaturverzeichnis

[Age] R. Ageloff. *Comprehensive Paradox 5 for Windows.*

[BG95] T. Biedassek and S. Gronemeier. Erweiterung von ML um graphische UI-Funktionen mit Tk. Interner Bericht, Universität Dortmund, Fachbereich Informatik, 1995.

[BJLL91] H.-J. Brede, N. Josuttis, S. Lemberg, and A. Lörke. *Programmieren mit OSF/Motif.* Addison-Wesley, 1991.

[CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer, Berlin [u.a.], 1981.

[DF89] E.-E. Doberkat and D. Fox. *Software Prototyping mit SETL.* B. G. Teubner, Stuttgart, 1989.

[DFG+92] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, and C. Pahl. ProSet – A Language for Prototyping with Sets Language Definition. Informatik-Bericht 02-92, Universität – GHS – Essen, Fachbereich Mathematik und Informatik, April 1992.

[Dil94] A. Diller. *Z – An Introduction to Formal Methods.* J. Wiley & Sons, Chichester, second edition, 1994.

[Dob92] E.-E. Doberkat et al. ProSet – A Language for Prototyping with Sets. Interner Bericht 15, 03/92, Universität – GHS – Essen, Fachbereich Mathematik und Informatik, 1992.

[EK92] E.Muller and B. Kalsow. *SRC Modula-3 V 2.11.* System Research Center, 1992.

[FGH+93] W. Franke, U. Gutenbeil, W. Hasselbring, C. Pahl, H.-G. Sobottka, and B. Sucrow. Prototyping mit Mengen – der ProSet-Ansatz. In H. Züllinghoven, W. Altmann, and E.-E. Doberkat, editors, *Requirements Engineering '93: Prototyping*, pages 165–174. B. G. Teubner. German Chapter of the ACM, Berichte 41, 1993.

[Fou90] Open Software Foundation, editor. *OSF/Motif Style Guide.* Prentice Hall International, 1990.

[Ger90] C. Gerety. A New Generation of Software Development Tools. *Hewlett-Packard Journal*, 41(3):48–58, June 1990.

[GKKZ92] K. Gottheil, H. J. Kaufmann, T. Kern, and R. Zhao. *X und Motif.* Springer-Verlag, 1992.

[Her94] M. Herczeg. *Software-Ergonomie, Grundlagen der Mensch-Computer-Kommunikation.* Addison-Wesley, 1994.

[HH92] T. R. Henry and S. E. Hudson. End User Controlled Interfaces: Creating Multiple View Interfaces for Data-Rich Applications. Technical report 92-04, University of Arizona, 1992.

[HS87] K. Haviland and B. Salama. *Unix System Programming.* International Computer Science Series. Addison-Wesley, 1987.

[HU93] J. E. Hopcroft and J. D. Ullmann. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie.* Addison-Wesley, 1993.

[Jac86] R. J. K. Jacob. A Specification Language for Direct-Manipulation User Interfaces. *ACM Transactions on Graphics*, 5(4):283–317, 1986.

[Kap95] C. Kappert. Integration von Persistenzkonzepten in eine Prototypingsprache und Realisierung mit Hilfe eines Nicht-Standard-Datenbanksystems. Master's thesis, Universität Dortmund, 1995.

[LM95] O. Lorenz and G. Monagan. Automatisches Indexieren von Liniengrafiken. In R. Kuhlen and M. Rittberger, editors, *Hypertext – Information Retrieval Multimedia.* Universitätsverlag Konstanz, 1995.

[Lor95] T. Lord. An Anatomy of Guile – The Interface to Tcl/Tk. Tcl/Tk Workshop, USENIX-Association, 1995.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Prentice Hall International, 1988.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Rae84] G. Raeder. *Programming in Pictures.* PhD thesis, University of Southern California, 1984.

[Sch85] P. Schefe. *Informatik – Eine konstruktive Einführung.* BI Wissenschaftsverlag, 1985.

[See90] J. Seekamp. Algebraische Spezifikation abstrakter Datentypen. Kapitel 3 eines internen Berichts der Projektgruppe WISDOM. Universität Dortmund, Fachbereich Informatik, Lehrstuhl für Software-Technologie, 1990.

[Smi91] J. D. Smith. *Object oriented programming with the X-Window System toolkits.* Wiley professional computing. Wiley, 1991.

[Sta94] C. Stary. *Interaktive Systeme – Software-Entwicklung und Software-Ergonomie.* Vieweg, 1994.

[Sun93] Sun Microsystems, Inc. *ToolTalk 1.1.1 Reference Manual*, 1993.

[Tan90] A. S. Tanenbaum. *Betriebssysteme.* Carl Hanser Verlag, 1990.

[Wik87] A. Wikstroem. *Functional programming using standard ML.* Prentice Hall, London, 1987.

[WS93] L. Wall and R.L. Schwartz. *Programmieren in Perl.* Ein Nutshell-Handbuch. Hanser, München [u.a.], 1993.

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang A

Graphische Dialoge

Im folgenden sind nun sämtliche Dialogfenster der in Kapitel 5 beschriebenen Dialoge abgebildet. Zur Orientierung befindet sich unter den Abbildungen jeweils in Klammern der Name des zum Dialog gehörenden Formulars.

Wie in den Abbildungen zu sehen ist, enthalten die Titelleisten der Dialogfenster eine ganzzahlige Ziffer, die auch in der Titelleiste des Pinboards wiederzufinden ist. Hierbei handelt es sich um den in Kapitel 7.2.1 und 7.2.2 beschriebenen Identifikator des Pinboards. Er wurde zusätzlich in die Titelleiste aufgenommen, um eine Gruppierung der zu einem Programm gehörenden Dialogfenster zu visualisieren. Insbesondere bei parallel gestarteten Programmen mit graphischer Ein- und Ausgabe ist der Benutzer so in der Lage, sämtliche erzeugten Fenster der jeweiligen Applikation zuzuordnen.

A.1 Graphische Dialoge zur Eingabe von ProSet-Werten

Abbildung A.1: Standard-Dialog zur Eingabe eines ProSet-Wertes (Standard-Eingabe)

Abbildung A.2: Dialog zur Eingabe einer ganzen Zahl (Schieberegler-Eingabe)

Abbildung A.3: Dialog zur Eingabe einer ganzen Zahl (Integer-Zehner-Block-Eingabe)

Abbildung A.4: Dialog zur Eingabe einer reellen Zahl (Bruch-Eingabe)

Abbildung A.5: Dialog zur Eingabe einer reellen Zahl (Real-Zehner-Block-Eingabe)

Abbildung A.6: Dialog zur Eingabe eines Dateinamens (Filename-Eingabe)

Abbildung A.7: Dialog zur Eingabe eines beliebigen ProSet-Wertes (Wahl-Eingabe)

Abbildung A.8: Dialog zur Eingabe eines Booleschen Wertes (True-oder-False-Eingabe)

Abbildung A.9: Dialog zur Eingabe eines Booleschen Wertes (Antwort-Eingabe)

Abbildung A.10: Dialog zur Eingabe einer Menge

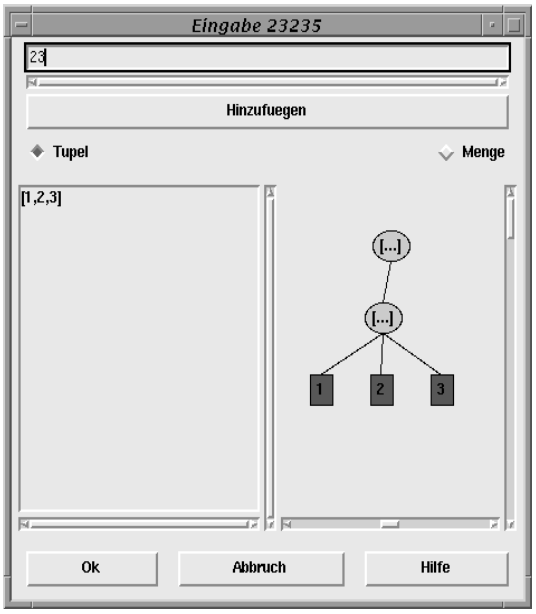


Abbildung A.11: Dialog zur Eingabe eines Tupels bzw. einer Menge

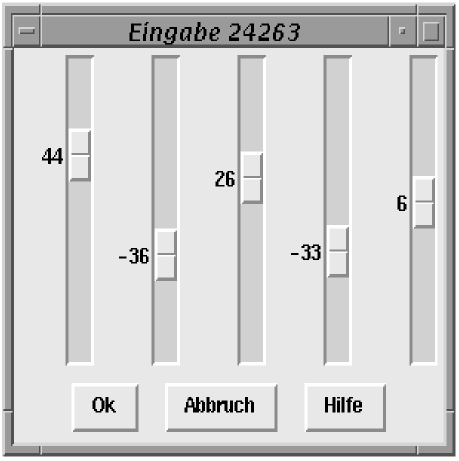


Abbildung A.12: Dialog zur Eingabe eines Tupels (Equalizer-Eingabe)

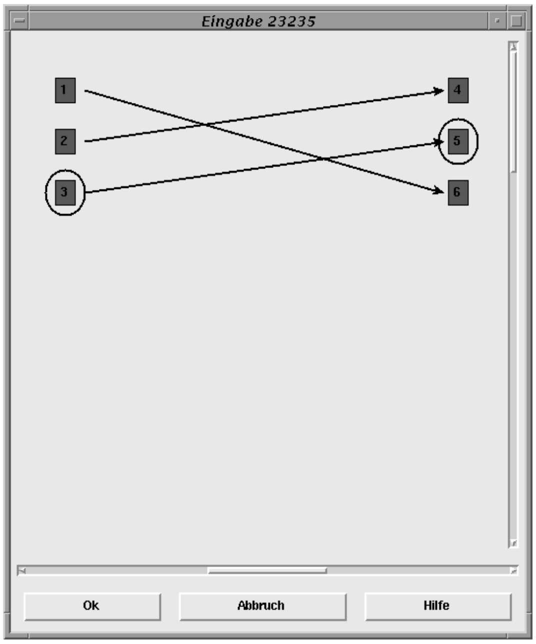


Abbildung A.13: Dialog zur Eingabe einer Abbildung (Map-Eingabe)

A.2 Graphische Dialoge zur Ausgabe von ProSet-Werten

Abbildung A.14: Standard-Dialog zur Ausgabe eines ProSet-Wertes (Standard-Ausgabe)

Abbildung A.15: Dialog zur Ausgabe einer ganzen Zahl (Wahl-Ausgabe)

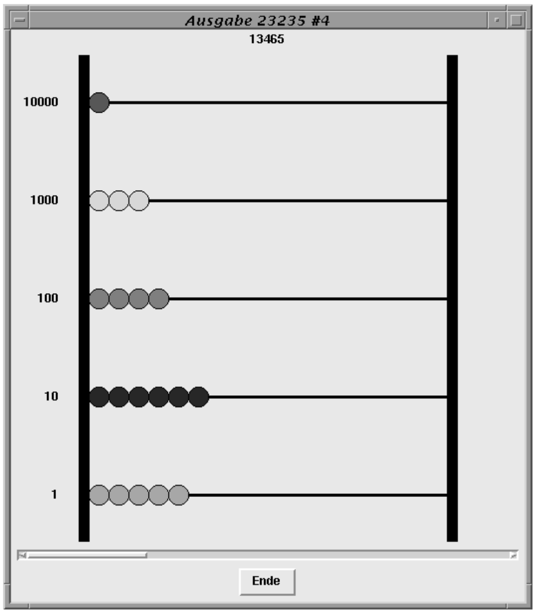


Abbildung A.16: Dialog zur Ausgabe einer ganzen Zahl (Abakus-Ausgabe)

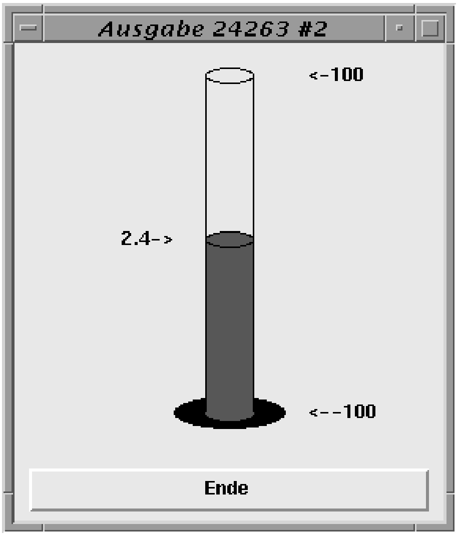


Abbildung A.17: Dialog zur Ausgabe einer reellen Zahl (Säulen-Ausgabe)

Abbildung A.18: Dialog zur Ausgabe einer ganzen bzw. reellen Zahl (Segment-Ausgabe)

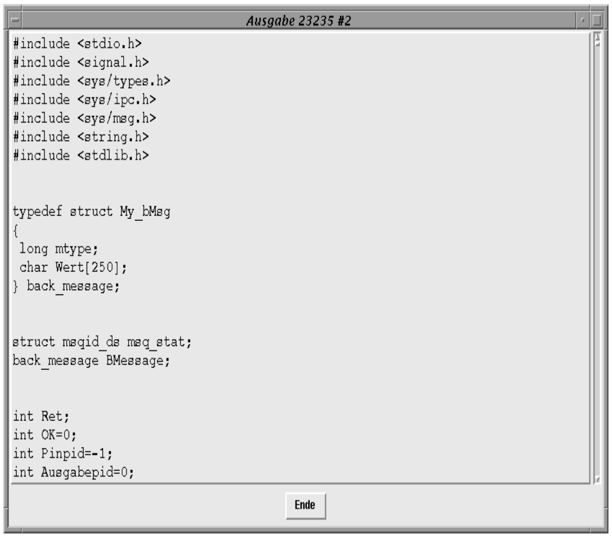


Abbildung A.19: Dialog zur Ausgabe einer Datei (File-Ausgabe)

Abbildung A.20: Dialog zur Ausgabe eines Nachricht (Message-Ausgabe)

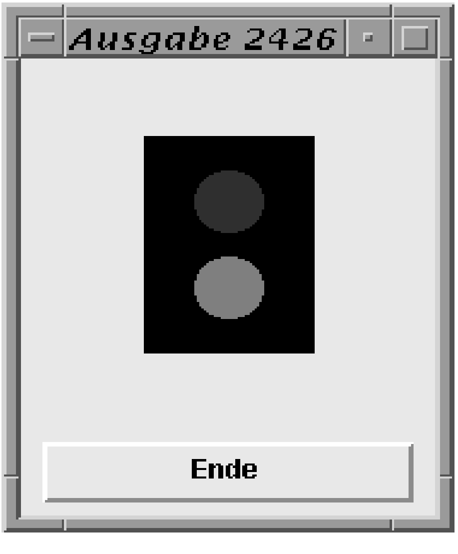


Abbildung A.21: Dialog zur Ausgabe eines Booleschen Wertes (Ampel-Ausgabe)

Abbildung A.22: Dialog zur Ausgabe eines Booleschen Wertes (Smiley-Ausgabe)

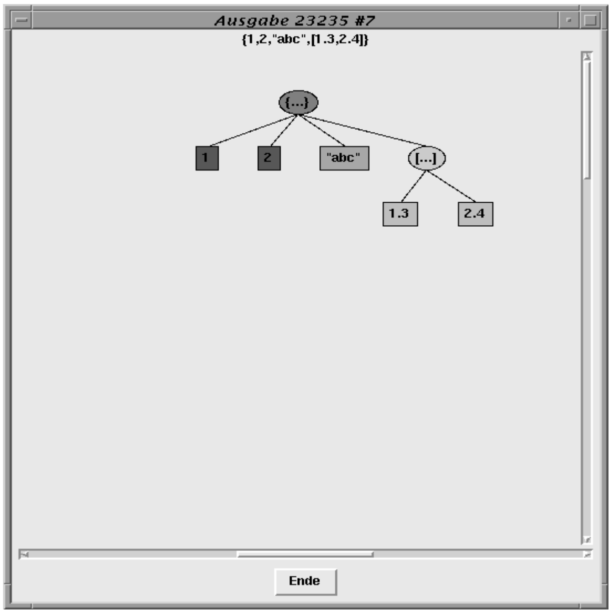


Abbildung A.23: Dialog zur Ausgabe eines Tupels bzw. einer Menge (Grafik-Ausgabe)

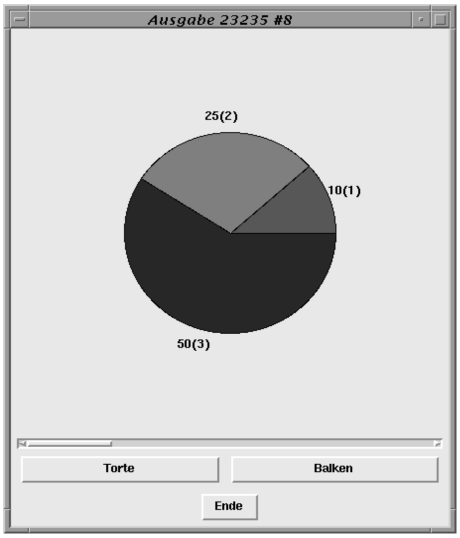


Abbildung A.24: Dialog zur Ausgabe eines Tupels bzw. einer Menge (Torten-Ausgabe)

Abbildung A.25: Dialog zur Ausgabe eines Tupels bzw. einer Menge (Element-Ausgabe)

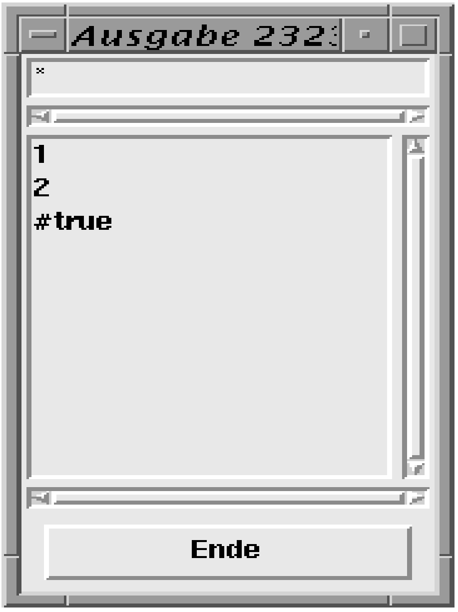


Abbildung A.26: Dialog zur Ausgabe eines Tupels bzw. Menge (Filter-Ausgabe)

Anhang B

Beispielhafte Konstruktion eines Dialoges

Dieser Anhang beschreibt die Erweiterung der ProSet-Bibliothek zur graphischen Ein- und Ausgabe von ProSet-Werten. Dabei gehen wir nach der in Kapitel 7.12 beschriebenen Weise vor und erweitern die ProSet-Bibliothek um einen Dialog zur Eingabe eines Datums vom Typ `Integer`. Die Eingabe erfolgt mittels eines Schiebereglers mit den Grenzen `Von` und `Bis`.

B.1 Erweiterung der C-Bibliothek

Die hier beschriebene Komponente der C-Bibliothek entspricht im wesentlichen der in Kapitel 7.1.1 beschriebenen Komponente (`Regular-Eingabe`). Auf eine erneute Beschreibung der einzelnen Anweisungen soll hier verzichtet werden. Diejenigen Programmzeilen, die geändert wurden, sind mit einem`(#)` am Ende der jeweiligen Programmzeile markiert.

Die C-Bibliothek `Xio.c` wird um folgende Komponente ergänzt:

```
char* XGetIntScale(Von,Bis)                                 (#)
int Von,Bis;                                                (#)
{
 char *Ergebnis;
 int Pid;         /*Prozessidentifikator des erzeugten
                   Kindprozesses*/
 char *Msg;       /*MessageQueue-Identifikator*/
 char *Show;      /*Show-Parameter*/
 char *Attribut1; /*Darstellungsattribut (Von)*/            (#)
 char *Attribut2; /*Darstellungsattribut (Bis)*/            (#)

 signal(SIGINT,Abbruch);

 Msg = mymalloc(100);
 Show = mymalloc(100);
 Attribut1 = mymalloc(100);                                 (#)
 Attribut2 = mymalloc(100);                                 (#)
```

```
if (MsgQueueID == -1)
   MsgQueueID = msgget(IPC_PRIVATE,0640);
sprintf(Msg,"%d",MsgQueueID);
sprintf(Show,"%d",0);
sprintf(Attribut1,"%d",Von);                               (#)
sprintf(Attribut2,"%d",Bis);                               (#)

if (CheckPath() == 0)
{
 Ergebnis = (char*)malloc(strlen("NoPath")+1);
 sprintf(Ergebnis,"%s","NoPath");
 free(Msg);
 free(Show);
 free(Attribut1);                                          (#)
 free(Attribut2);                                          (#)
 return Ergebnis;
}
if (Pinpid == -1)
{
  SetPinboardID();
  Pinpid = fork();
}
if (Pinpid == 0) /*Code des Kindprozesses*/
{
 StartPinboard();
}
else /*Code des Vaterprozesses*/
{
 pid = fork();
 if (pid == 0) /*Code des Kindprozesses*/
 {
  char* Pfad;
  Pfad = (char*)malloc(strlen(getenv("INP"))
                       +strlen("\Input")+1);
  sprintf(Pfad,"%s/Input",getenv("INP"));
  execl(Pfad,"Input",Msg,Show,"IntScEingabe",              (#)
        Attribut1,Attribut2,NULL);                         (#)
 }
 else /*Code des Vaterprozesses*/
 {
  free(Msg);
  free(Show);
  free(Attribut1);                                         (#)
  free(Attribut2);                                         (#)
  BMessage.mtype = 0;
  while (BMessage.mtype == 0)
  {
```

```
     ReturnVal = msgrcv(MsgQueueID,&BMessage,250,-5,0);
    }
    switch (BMessage.mtype) {
    case 1: Ergebnis = (char*)malloc(250);
            strcpy(Ergebnis,&(BMessage.Wert[0]));
            return Ergebnis;
            break;
    case 2: Ergebnis = (char*)malloc(strlen("NoDisplay")+1);
            sprintf(Ergebnis,"%s","NoDisplay");
            return Ergebnis;break;
    case 3: Ergebnis = (char*)malloc(strlen("Close")+1);
            sprintf(Ergebnis,"%s","Close");
            return Ergebnis;break;
    case 4: Ergebnis = (char*)malloc(strlen("Cancel")+1);
            sprintf(Ergebnis,"%s","Cancel");
            return Ergebnis;break;
    default:
    }
   }
  }
 }
```

Die Änderungen in der oben abgebildeten C-Bibliotheks-Komponente betreffen im wesentlichen den Aufruf von `Input` zusammen mit den zusätzlichen Parametern `Attribut1` und `Attribut2`.

B.2 Erzeugung des zugehörigen Tcl-Skriptes

Auch hier wurde das in Kapitel 7.6.1 beschriebene Tcl-Skript zur Eingabe eines ProSet-Wertes (Standard-Eingabe) übernommen und entsprechend abgeändert.

```
set libpath $env(TCL_BIBS)
set auto_path [linsert $auto_path 0 $libpath]
set x "Pinboard $PinID";

proc Insert Ins {
 set Trans [UnTransform $Ins];
 .scl set $Trans                                       (#)
}
```

Die Funktion `Insert` fügt das mit der Variablen `Ins` übergebene Datum in das Formular des Dialogfensters ein. Dabei muß der Schieberegler `.scl` auf den entsprechenden Wert eingestellt werden.

```
scale .scl -from $Argv1 -to $Argv2 -length 10c \       (#)
           -orient horizontal -bigincrement 0 \        (#)
           -digits 0                                   (#)
```

Mit dieser Anweisung wird ein Schieberegler mit den Grenzen `Argv1` und `Argv2` erzeugt. `Argv1` und `Argv2` enthalten die dem Eingabeprozeß `Input` übergebenen Parameter `Attribut1` und `Attribut2`, die wiederum die Parameter `Von` und `Bis` der zugehörigen C-Funktion enthalten (siehe dazu 7.2.1).

```
button .btcancel -relief raised \
                 -text "Abbruch" \
                 -command \
                       {
                         if {[lsearch [winfo interps] $x] != -1} {
                                 send $x {set SetInp 0;fillall}
                         }
                         Sende 4;
                         exit
                       }


button .help -relief raised \
             -text "Hilfe" \
             -command {
                       Help "IntScEingabe" $Argv1 $Argv2 (#)
                      }
```

Diese Änderung betrifft die Marke des darzustellenden Hilfetextes. Zusätzlich werden die Grenzen des Schiebereglers als Parameter übergeben.

```
if {$Show == 0} {
 button .btok -text "Ok"\
              -command \
                  {
                   set Input [.scl get];                (#)
                   set Trans [Transform $Input];
                   set Attribs "$Argv1,$Argv2"          (#)
                   if {[lsearch [winfo interps] $x] != -1} {
                          send $x finsert $Trans \      (#)
                                          Schieberegler-Eingabe \ (#)
                                          $Attribs;     (#)
                          send $x {set SetInp 0;fillall}
                   }
                   Sende 1
                   exit
                  }
}
```

Durch Drücken des `Ok`-Buttons wird der mittels des Schiebereglers eingegebene Wert zusammen mit dem Namen des Formulars (`Schieberegler-Eingabe`) und den Grenzen des Schiebereglers als Darstellungsparameter in das Pinboard eingefügt.

```
if {$Show == 1} {
 button .btok -text "Ende" -command "exit"
}


pack .scl                                                (#)
pack .btok -side bottom -side left -padx 2m \
           -pady 1m -expand 1 -fill x
pack .btcancel -side bottom -side left -padx 2m \
               -pady 1m -expand 1 -fill x
pack .help -side bottom -side left -padx 2m \
             -pady 1m -expand 1 -fill x


if {$Show == 0} {
 if {[lsearch [winfo interps] $x] != -1} {
              send $x fillIntSc $Argv1 $Argv2            (#)
              send $x {set SetInp 1}
 }
```

Die hier vorgenommenen Änderungen betreffen im wesentlichen die Geometrie des Dialoges sowie den dialogabhängigen Filter, der im Pinboard aktiviert wird (`fillintsc`).

```
}
if {$Show == 1} {
 wm title . "Alte Eingabe $PinID"
 .scl set $Input                                         (#)
 .scl config -state disabled                             (#)
 .help config -state disabled
 .btcancel config -state disabled
}
```

Zur Reproduktion des Dialoges muß der mit `Input` übergebene Wert mit Hilfe des Schiebereglers eingestellt werden.

B.3 Anpassung des Pinboards

Um den oben beschriebenen Dialog reproduzieren zu können, muß die Tcl-Prozedur `show` innerhalb der Tcl-Bibliothek `user.tcl` um einen entsprechenden Fall innerhalb der `switch`-Anweisung ergänzt werden.

```
proc show {} {
 global list ShowPidList PinID
 global env;
 set OPath $env(OUT)
 set IPath $env(INP)
 if {[.files curselection] != ""} {
```

```
     set Input [.files curselection]
     foreach elem $Input {
      set Trans [.files get $elem]
      set Param $Trans;
      set Dummy "\[0-9\]"
      set Typ [.types get $elem]

      switch $Typ {
        Standard-Eingabe {
         set Pid [exec $IPath/Input $PinID 0 1 "StringEingabe.tcl"\
                                    $Param &]
        }

        Schieberegler-Eingabe {                                  (#)
         set grenzen [Parse "\{[.param get $elem]\}"];           (#)
         set Pid [exec $IPath/Input $PinID 0 1 "IntScEingabe" \  (#)
                                    [lindex $grenzen 0] \        (#)
                                    [lindex $grenzen 1] \        (#)
                                    $Param &]                    (#)
        }                                                        (#)
        Standard-Ausgabe {
         set Pid [exec $OPath/Output $PinID 0 1 $Param \
                                     "StringAusgabe.tcl" &]
        }
      }
      set ShowPidList [lappend ShowPidList $Pid]
     }
 }
}
```

Abhängig vom Formular des Dialoges (`Schieberegler-Eingabe`) wird zunächst mittels der Funktion `Parse` eine Liste der Darstellungsattribute des anzuzeigenden Dialoges erzeugt. Die Elemente dieser Liste dienen als Parameter des Eingabeprozesses `Input`, der mit Hilfe des Tcl-Kommandos `exec` erzeugt wird. Das darzustellende Datum, welches im Pinboard selektiert wurde, befindet sich in der Variablen `Param`. Diese muß als **letzter** Parameter dem Eingabeprozeß `Input` übergeben werden.

Ein Filter hat die Aufgabe, sämtliche vom Pinboard aufgezeichneten Daten abhängig vom aktuellen Eingabe-Dialog zu filtern. Um die für diesen Dialog in Frage kommenden Daten zu filtern, wird die Tcl-Bibliothek `filter.tcl` um folgenden Filter ergänzt:

```
proc fillIntSc {f t} {                                   (#)
 global list WhatFilter
 global From To                                          (#)
 set WhatFilter 7;                                       (#)
 set From $f;                                            (#)
 set To $t                                               (#)
 .files delete 0 end;
```

```
 .types delete 0 end;
 .param delete 0 end;
 foreach ele $list {
  set Wert [lindex $ele 0]
  if {($Wert<=$t)&&($Wert>=$f)&&\                     (#)
      ([regexp {^(-)?[0-9]+$} $Wert])} {Einfuegen $ele}  (#)
 }
 .files selection clear 0 end;
 .files selection set end end
}
```

Dieser Filter filtert sämtliche aufgezeichneten Daten, die einem ganzzahligen Wert entsprechen und zwischen `f` und `t` liegen. Die aufgezeichneten Daten befinden sich in der globalen Variablen `list`. Da der aktivierte Filter bekannt sein muß, werden die globalen Variablen `WhatFilter`, sowie die zugehörigen Filterparameter `From` und `To` mit entsprechenden Werten belegt. `WhatFilter` bezeichnet einen Filteridentifikator, wobei darauf zu achten ist, daß innerhalb der Tcl-Bibliothek jedem Filter ein eindeutiger Identifikator zugewiesen wird. Nachdem sämtliche dargestellten Daten des Pinboards zunächst entfernt werden, werden mittels der Funktion `Einfuegen` sämtliche Daten innerhalb der Tcl-Liste `list` filterabhängig in dem Pinboard dargestellt.

Da bei einer Eingabeaufforderung persistente Daten vom Benutzer geladen werden können, die abhängig vom aktuellen Eingabe-Dialog gefiltert werden müssen, wird die Prozedur `Laden` in der Tcl-Bibliothek `user.tcl` folgendermaßen abgeändert:

```
proc Laden {} {
 global PFile WhatFilter FReg FAnz FillSet1
 global FillSet2 FVor FNach SetInp
 global From To                                        (#)
 PFileEingabe "Bitte geben Sie einen Bezeichner fuer \
              die zu ladenden Daten an"
 Anfrage
 if {$SetInp==1} {
        switch $WhatFilter {
        0 {fillall}
        1 {fillEq $FAnz $From $To}
        2 {fillmap $FillSet1 $FillSet2}
        3 {filldef}
        4 {filltupset}
        5 {fillRegMenge $FReg}
        6 {fillReg $FReg}
        7 {fillIntSc $From $To}                        (#)
        8 {fillBool}
        9 {fillRealBl $FVor $FNach}
        10 {fillfile}
        11 {fillIntBl}
        12 {fillReal}
        13 {nofill}
    }
```

```
 }
}
```

B.4 Erweiterung des Hilfetextes

Die Hilfedatei `PSXio.hlp` wird folgendermaßen ergänzt:

```
begin<IntScEingabe.ger>
Legen Sie nun mit Hilfe des Schiebereglers eine ganze Zahl fest.
Die untere Grenze betraegt <arg1>. Die obere Grenze betraegt <arg2>.
Durch Druecken des Ok-Buttons beenden Sie Ihre Eingabe.
Durch Druecken des Abbruch-Buttons brechen Sie die Eingabe ab.
end<IntScEingabe.ger>
begin<IntScEingabe.eng>
By moving the slider you are able to specify an integer as input.
The range of possible inputs lies between <arg1> and <arg2>.
end<IntScEingabe.eng>
```

B.5 Erweiterung der ProSet-Bibliothek

Abschließend muß die ProSet-Bibliothek um eine weitere Komponente ergänzt werden. Beispielsweise könnte folgende Komponente implementiert werden:

```
procedure PS_XGetIntScale(rd von, rd bis);
persistent constant scan : "StdLib";
begin
 if (type(von)/=integer) or (type(bis)/=integer) then
  signal xio_type_mismatch();
 else if (von>=bis) then escape xillegal_operand();
      else
       Datum:=c_fct_call XGetIntSc(von:c_integer,bis:c_integer) c_string;
       if Datum="NoPath" then
        escape NoPath();
       else if Datum="NoDisplay" then
             escape NoDisplay();
            else if Datum="Cancel" then
                  escape Cancel();
                 else if Datum="Close" then
                       escape Close();
                      else
                       return scan(Datum);
                      end if;
                 end if;
            end if;
       end if;
      end if;
```

```
 end if;
end PS_XGetIntScale;
```

Auf eine Erweiterung der Dialoge zur *Ausgabe* von ProSet-Werten wollen wir hier nicht mehr eingehen. Sie deckt sich im wesentlichen mit der oben beschriebenen Erweiterung.