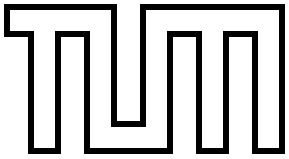


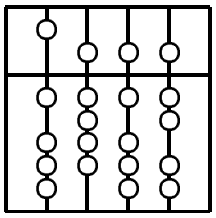
TECHNISCHE
UNIVERSITÄT
MÜNCHEN

Fakultät Für Informatik

INVITING

InterViews Tutorial Introduction Guide
for beginners

München, den 22. Dezember 1993



Vorwort

Graphische Benutzeroberflächen (**GUI**[1]) treten mehr und mehr in den Vordergrund, wenn es darum geht, die Schnittstelle zwischen Mensch und Maschine, bzw Computer, zu verbessern. Standardisierte Menues und Benutzerführungen geben selbst dem unerfahrenen User die Möglichkeit, sich in kurzer Zeit in neuen, komplexen Programmsystemen zurechtzufinden. Die zunehmende Verbreitung von schnelleren Grafikkarten ermöglicht gleichzeitig die verbesserte technische Umsetzung dieser Oberflächen, indem die „lags“ zwischen Eingabe und Ausführung eines Befehls auf so minimale Zeitspannen reduziert werden, daß auch in zunehmenden Maßen Programmierer und erfahrene Benutzer auf diese Oberflächen zurückgreifen.

Mit *InterViews* wurde von der Stanford University ein Tool entwickelt, mit dem es nicht nur für Workstation– (X für Unix) sondern auch in PC – Betriebssystemen (Linux, evtl. Windows NT) sowie vielen anderen Architekturen und Systemen möglich ist, objektorientierte **GUI**'s in C++ zu schreiben, indem es eine erweiterbare Grundbibliothek an Klassen und Funktionen bereitstellt, die die Programmierung erheblich vereinfachen. Insbesondere die leichte Portierbarkeit zwischen oben genannten Architekturen ist eine der herausragensten Eigenschaften von *InterViews*.

Die bisherige Dokumentation erweist sich für den Anfänger als leider nicht sehr hilfreich. Selbst der etwas erfahrenere Programmierer (dazu zähle ich mich) hat doch einige Probleme, mit dem mitgelieferten Manual [MAN3.1] zurechtzukommen. Dieses Problem ergab sich schon bei der Version 3.0, weshalb auch hierzu das sehr gute Manual „Einführung in Interviews“ von Stefan Mayer ([IV3.0]) entworfen wurde, dessen Studium es mir ermöglichte, mich mit *InterViews* vertraut zu machen.

Bei dem Umstieg auf Version 3.1 fiel es mir selbst mit meinen Grundkenntnissen und etwas Erfahrung sehr schwer, mich in den neuen Strukturen und Klassendefinitionen zurechtzufinden. Das Manual half in den meisten Fällen durch seine doch etwas sehr kurzgeratenen Beschreibungen nicht oder nur durch mehrmaliges Durchlesen des gesamten Textes. Deswegen wurde der Entschluß gefaßt, diese neue Einführung für die *InterViews* – Version 3.1 anzufertigen, die sowohl

[1] GUI bedeutet *graphical user interface*, soviel wie graphische Benutzerschnittstelle.

die guten Eigenschaften der alten Einführung beibehält, als auch Fehler beseitigt, und die mit den neuen Strukturen und Klassendefinitionen konform ist.

Die einfachste Art und Weise, mit *InterViews* zurechtzukommen, ist und bleibt das Studium von Beispielprogrammen, was ich auch dem Anwender weiterhin wärmstens empfehlen möchte. Die mit dem *InterViews* – Kit mitgelieferten „examples“ liefern sehr schöne Beispiele; wenn auch oft zu viele verschiedene neue Aspekte in ein einziges Programm gepackt wurden, weshalb die Hauptaufgabe darin besteht, diese zu entwirren und die einfacheren Programmstücke zum besseren Verständnis daraus zu extrahieren.

Des weiteren ist die Newsgroup „comp.windows.interviews“ sehr zu empfehlen, hier werden Probleme der Compilation und Programmierung besprochen, und auch oft gelöst. Man sollte jedoch beachten, das viele gepostete Fragen oft sogar von einem der Autoren selbst, Mark Linton, per email beantwortet werden und daher leider nicht in der Newsgroup als posting erscheinen, was zu der fälschlichen Annahme führen könnte, diese Fragen blieben unbearbeitet.

Die vorliegende Arbeit basiert auf der Arbeit „Einführung in InterViews 3.0“ von Stefan Mayer ([IV3.0]), dem ich auf diesem Wege nochmals Dank sagen möchte. Für etwaige Fehler oder Verbesserungsvorschläge bin ich jederzeit offen und dankbar und unter der email Adresse
kuerten@informatik.tu-muenchen.de sogar erreichbar :) .
(Zumindest solange mein Studium noch andauert)

München, den 22. Dezember 1993 Thomas Kürten.

Inhaltsverzeichnis

Kapitel 1

Einführung

In diesem Abschnitt gehe ich kurz darauf ein, welche Grundkenntnisse für das Verständnis dieses Tutorials unabdingbar sind. Einige zentrale Klassen werden behandelt und anschließend anhand eines einfachen Beispiels erläutert.

1.1 Voraussetzungen

Zum Verständnis von *InterViews* ist es unerläßlich, sich zumindest mit der Programmiersprache C++ gut auszukennen; Grundkenntnisse in objektorientierter Programmierung sind auch von großem Vorteil. *InterViews* stellt eine Programmbibliothek zur Verfügung, die fast ausschließlich in C++ verfasst ist, und deren Funktionen durch Vererbung, Ableitung, und andere C++ – spezifische Konstrukte nutzbar gemacht werden können.

Weiterhin sollte der Leser auch mit seinem Computersytem umgehen können. Meist sind an Universitäten Unix–Betriebssysteme verbreitet, die von einem Systemverwalter betreut werden. Wenn *InterViews* noch nicht auf Ihrem System installiert ist, sollten Sie sich lieber an diesen wenden, da die Installation sich nicht unbedingt einfach gestaltet.

Ein wenig Erfahrung im Umgang mit graphischen Oberflächen ist notwendig; Begriffe wie „Maus“, „Fenster“, oder „Menü“ sollten Ihnen schon bekannt sein.

Falls Sie auf einem dieser Gebiete größere Hindernisse sehen, dann machen Sie sich bitte vorher damit vertraut, im folgenden wird hier nicht weiter darauf eingegangen.

1.2 Aufbau von InterViews

1.2.1 Überblick

InterViews ist ein Softwaresystem für auf Fenstern basierende Applikationen. Die Intention von *InterViews* liegt in der objektorientierten Handhabung von Fenstern, Buttons, Menüs, Fensterinhalten und ähnlichem.

Dabei stellt *InterViews* eine Anzahl von Klassen zur Verfügung, die das Verhalten von interaktiven Objekten definieren. Diese lassen sich in zwei Hauptkategorien aufteilen: Die *Protocols* und die *Kits*.

Ein *Protocol* definiert eine Anzahl von Operationen die ein Objekt ausführen kann (z.B. Zeichnen oder verarbeiten von Eingaben) und ein *Kit* definiert eine Menge an Funktionen, mit denen andere Objekte entworfen werden können (z.B. **WidgetKit** oder **LayoutKit**).

Mit diesen Hilfsmitteln läßt sich eine Menge an Arbeitsaufwand sparen und der Programmierer kann sich auf seine Hauptalgorithmen und schwierigeren Problemstellungen konzentrieren.

1.2.2 Die Headerdateien

Da *InterViews* eine Objektorientierte Programmbibliothek darstellt, sind natürlich die wichtigsten Bestandteile in Headerfiles zu finden, die am Anfang des Programms (mit **#include**) eingebunden werden. Wer sich in seinem System auskennt kann sich in den verschiedenen **include** – Subdirectories der *InterViews* – Applikation einmal umsehen und sollte dies auch tun. Für den Anfänger sind sicher die Unterdirectories **InterViews** und **IV–look** von Interesse. Dabei enthält **InterViews** eigentlich alle Grundklassen–Header und **IV–look** die Headerdateien **kit.h** (für die Klassen **WidgetKit** und **LayoutKit**), sowie **dialogs.h** und **menu.h**, die in den nachfolgenden Kapiteln noch ausführlicher besprochen werden.

Normalerweise sollte gelten: Für jede Klasse ein eigenes Headerfile („**.h**“) für die Deklaration der Klasse und ein eigenes „**.c**“ – Sourcecodefile für die Implementation; dies sollten Sie auch bei der Entwicklung von eigenen Programmen beherzigen.

Spätestens wenn Sie schon einmal programmierte Objekte wiederverwenden wollen, werden Sie froh sein, daß Sie sich nicht erst alles zusammensuchen müssen. Leider haben die Erfinder von *InterViews* anscheinend nicht sehr viel von dieser Technik gehalten. Das Ergebnis dürfen Sie jederzeit im **examples** – Unterverzeichnis des *InterViews* – Pakets begutachten. Die Übersichtlichkeit dieser Programme spottet jeder Beschreibung und der Aufbau ist wahrscheinlich nur noch dem Programmierer selbst ersichtlich; wer sich einmal daran wagt, einzelne Teile daraus zu verwenden, wird mir sicher beipflichten. Dabei sind gute Programmteile durchaus enthalten, und der fortgeschrittene *InterViews* – Benutzer kommt

gar nicht darum herum, sich mit diesen Beispielen auseinanderzusetzen, da die Dokumentation [MAN3.1] nicht gerade sehr hilfreich ist.

1.3 Grundlegende Definitionen und Klassen

In diesem Unterabschnitt werden Definitionen und Klassen vorgestellt, die eine zentrale Bedeutung in *InterViews* haben, und daher meist schon nach Einbinden eines beliebigen Headerfiles zur Verfügung stehen.

Grundsätzlich sind die Klassen in *InterViews* ähnlich der C++ „Class“ – Definition zu verstehen. Da aber ein gewisser Unterschied besteht (der uns nicht weiter interessieren muß), werden die meisten Klassen in *InterViews* als „Interface“ deklariert.

Wenn Sie das erste Beispiel jetzt sofort ausprobieren wollen, dann folgen Sie bitte den Anweisungen in Abschnitt 1.4 auf Seite 15. Im folgenden möchte ich zuerst die Grundlagen erklären.

1.3.1 Boolean

Der Typ **boolean** umfaßt die Konstanten *TRUE* und *FALSE*, und wird wie in herkömmlichen Programmiersprachen (wie z.B. *Pascal* oder *Modula*) aufgefaßt.

1.3.2 Coord

Viele Positionsangaben, die die Ausgabe am Bildschirm betreffen, werden in **Coord**s ausgedrückt. Der Typ **Coord** wird in *InterViews* durch eine float–Zahl repräsentiert, die zur besseren Realisierung auf dem Bildschirm vor der Ausgabe gerundet wird.

1.3.3 Resource

Da C++ keine sog. „*garbage collection*“[1] anbietet, ist es notwendig, den Programmstack auf andere Art und Weise von nicht mehr benötigten Zeigern und Allokierungen zu bereinigen. Dies wird in *InterViews* mit der Klasse **Resource** ermöglicht. Fast alle Objekte in *InterViews* sind von **Resource** abgeleitete Klassen, welche sich wie folgt darstellt:

```
interface Resource {
   static void ref(const Resource*);
   static void unref(const Resource*);
   ...
};
```

Jedes Objekt einer solchen Klasse enthält also einen sogenannten Referenzzähler, der mit Null initialisiert ist. `Resource::ref` erhöht den Zähler um eins und `Resource::unref` erniedrigt den Zähler um eins. Erreicht der Referenzzähler durch einen `Resource::unref` – Aufruf wieder den Wert Null, so wird das Objekt automatisch gelöscht. Besondere Beachtung findet diese Vorgehensweise bei sog. „*shared objects*“. Beispiele hierfür sind zum Beispiel **Color**, **Font**, **Brush** oder **Glyph**. Diese Klassen haben die Gemeinsamkeit, daß sie einmal definiert, immer wieder an verschiedenen Programmstellen benötigt werden, und daher auch nur einmal eingerichtet werden sollten. Deshalb werden diese Objekte *geshared*, also mit vielen Programmteilen *geteilt*, verhalten sich jedoch anders als eine *globale Variable*, die ja nicht mehr gelöscht werden kann.

Selber definierte Objekte, die Klassen dieser Kategorie enthalten, sollten daher die Verwendung dieser *shared objects* mit den zugehörigen Referenzzähler– Aufrufen klammern, da sie sonst womöglich gleich nach der Erstellung wieder gelöscht werden, was ein häufiger Grund für Systemabstürzen (*core dumps*) während des Programmablaufs bei schlecht konzipierten Programmen ist.

Die einzige Ausnahme für diese Regel tritt dann ein, wenn ein Programmstück „**P**“ ein *shared object* erzeugt und danach dieses nur an eine andere Funktion „**F**“ als Argument übergibt, und es sonst nicht mehr verwendet. In diesem Fall ist es nicht notwendig, die Stelle mit **ref(...)** und **unref(...)** zu klammern, da nach *InterViews* – Konvention die Funktion „**F**“ dafür verantwortlich ist, zu re- und dereferenzieren.

Bei selber geschriebenen Programmen sind oft *beide* Programmstücke („**P**“ und „**F**“) selber zu entwerfen, und oft ist diese Konvention nicht immer so einfach einzuhalten. Wichtig ist aber in jedem Fall, *daß* das Objekt in einem der Programmteile von Referenzzähleraufrufen geklammert wird.

[1] garbage collection = Müll aufsammeln

Hier noch drei kurze Beispielprogrammstücke, um zu verdeutlichen, wo nun der Aufruf von `Resource::ref` und `Resource::unref` explizit zu erfolgen hat:

```
Color *black = new Color(0.0, 0.0, 0.0);
canvas->fill(black);
```

Hier wird nicht geklammert; der zugehörige Funktionsteil von Canvas übernimmt das:

```
void Canvas::fill(Color *col) {
     Resource::ref(col);
     ...
     Resource::unref(col);
}
```

Bei folgenden Programmteil muß geklammert werden:

```
Color *black = new Color(0.0, 0.0, 0.0);
Resource::ref(black);
       .
       .   //Programmteil mit Verwendung von *black
       .
Resource::unref(black);
```

Hinweis: *Auf Keinen Fall dürfen shared objects mit „delete“ zerstört werden ! Außerdem werden diese Objekte nur mittels „new“ erzeugt und dürfen demzufolge nicht als automatische Variablen verwendet werden. Der explizite Aufruf von „delete“ auf ein shared object führt fast immer zwangsläufig zu einem Systemabsturz, da der Verwaltungsmechanismus gestört wird und dies Speicherzeiger in beliebige Adressen zur Folge hat.*

1.3.4 Glyph

Glyphs *sind die zentralen Objekte in InterViews zum Aufbau einer graphischen Benutzeroberfläche.*

Die Klasse **Glyph** ist eine abstrakte Basisklasse, d.h. es können nur Objekte abgeleiteter Klassen direkt erzeugt werden. Fast alle Klassen, die auf dem Bildschirm darstellbare Objekte erzeugen, sind von **Glyph** direkt oder indirekt abgeleitet. **Glyph** ist von **Resource** abgeleitet. D.h. alle von **Glyph** abgeleiteten Klassen sind *shared objects* und müssen demach womöglich referenziert werden (Siehe Abschnitt 1.3.3).

An dieser Stelle möchte ich eine Konvention einführen: Wenn im weiteren von „Glyphs“ gesprochen wird, so sind damit Objekte der Klasse **Glyph** *oder*

Objekte von Klassen gemeint, die von **Glyph** *abgeleitet sind. Außerdem wird in den Fällen, in denen Zeiger auf* **Glyph***–Objekte als Parameter angegeben werden müssen, häufig einfach von der „Übergabe eines Glyphs“ gesprochen.*
Man unterscheidet drei Arten von **Glyph**s:

1. Primitive **Glyphs** wie z.B. **Labels** für Text haben keine Teilkomponenten und werden direkt auf dem Bildschirm abgebildet. Sie stellen bei einer Glyphhierarchie die Blätter dar.

2. **MonoGlyphs** enthalten genau einen **Glyph**, der auch als „*Body*“ des **MonoGlyphs** bezeichnet wird. **MonoGlyphs** ändern das Aussehen bzw. das Verhalten ihres *Bodys*. So zeichnet ein **Background–MonoGlyph** beispielsweise einen farbigen Hintergrund unter seinen *Body*. Es gibt eine Vielzahl verfügbarer **MonoGlyphs**, die die unterschiedlichsten Aufgaben erfüllen können.

3. **PolyGlyphs** wie z.B. `LayoutKit::vbox` enthalten einen oder mehrere **Glyph**– Komponenten. Diese Komponenten werden auf bestimmte Weise angeordnet und auf dem Bildschirm abgebildet.

Allgemein werden *InterViews* – Benutzeroberflächen durch Konstruktion einer Glyphhierarchie aus **PolyGlyphs**, **MonoGlyphs** und primitiven **Glyphs** aufgebaut. Die Wurzel dieser Hierarchie wird einem **Window**–Objekt als Argument übergeben. Bei der Darstellung des betreffenden Bildschirmfensters wird die gesamte Hierarchie dann automatisch am Bildschirm abgebildet.

Eine Glyphhierarchie ist übrigens kein Baum, sondern ein azyklischer gerichteter Graph, weil ein Knoten mehrere Vorgänger haben kann. Dies beruht auf der Tatsache, daß **Glyphs** „*shared objects*“ sind. Aus diesem Grund muß beispielsweise ein **Label**, der in einer Dialogbox zweimal vorkommt, nur einmal erzeugt werden. Sehen wir uns zur Verdeutlichung Abbildung 1.1 an. Ein **PolyGlyph**

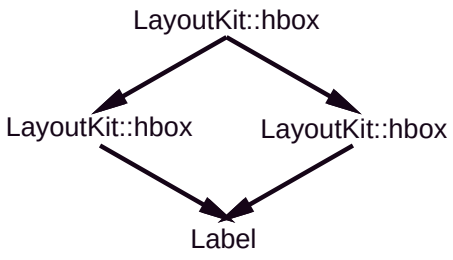


Abbildung 1.1: Beispiel einer einfachen Glyphhierarchie

(`LayoutKit::hbox`) soll wiederum zwei **PolyGlyphs** mit jeweils demselben **Label** enthalten. Der zughörige *InterViews* – Code sieht in etwa wie folgt aus:

```
WidgetKit& kit = *WidgetKit::instance();
const LayoutKit& layout = *LayoutKit::instance();
Label *l = kit.label("Hallo!");
PolyGlyph *b1 = layout.hbox(l);
PolyGlyph *b2 = layout.hbox(l);
PolyGlyph *b3 = layout.hbox(b1, b2);
```

(hierbei soll Sie die Tatsache, daß wir einige Teile dieses Programms noch nicht kennengelernt haben, nicht stören; Sie müssen zum Verständnis nur wissen, daß ein von `LayoutKit::vbox` generierter **PolyGlyph** die enthaltenen **Glyphs** – also einen oder mehrere – horizontal nebeneinander darstellt.)

Wird der Zeiger auf die Wurzel dieser Hierarchie, also „b3“, an ein **Window**–Objekt übergeben und das entsprechende Fenster am Bildschirm abgebildet, so enthält das Fenster im Prinzip das in Abbildung 1.2 angegebene Bild. Die schwar-

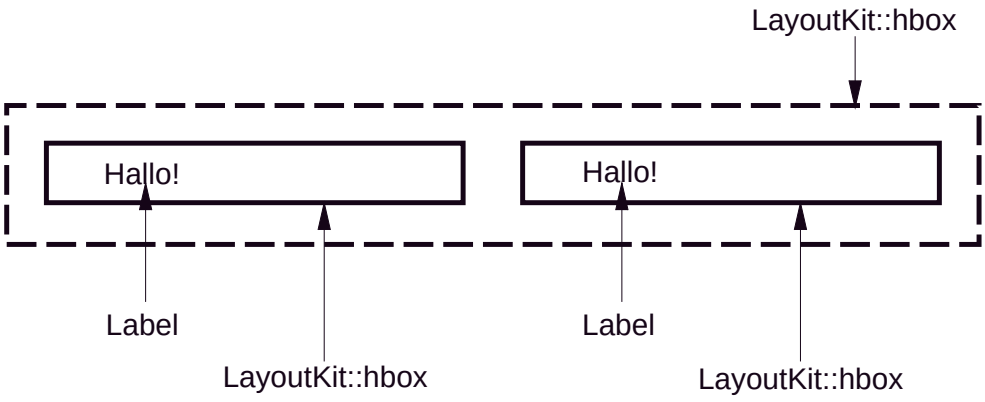


Abbildung 1.2: Stilisierte Darstellung der einfachen Glyphhierarchie

zen Linien um die Boxen und die Zwischenräume dienen hier nur Ihrer Orientierung und werden von *InterViews* selbstverständlich nicht dargestellt.

Das **Label** „l“ wurde also lediglich einmal erzeugt und hat nach Abbildung 1.1 zwei Vorgänger in der Glyphhierarchie. Durch dieses Konzept der *shared objects* kann unter Umständen eine erhebliche Speicherplatzersparnis erreicht werden. Im obigen Beispiel könnten übrigens auch die beiden **PolyGlyphs** „b1“ und „b2“, die den **Label** enthalten, als ein einziger „*shared* **PolyGlyph**“ realisiert werden.

1.3.4.1 Label

Mit Hilfe dieser Klasse werden Objekte erzeugt, die Strings am Bildschirm darstellen. Der wichtigste Teil der Deklaration hat folgende Gestalt:

```
class Label : public Glyph {
  public:
    Label(const char*, const Font*, const Color*);
    Label(const char*, int len, const Font*, const Color*);
};
```

Der Konstruktor erhält als ersten Parameter den darzustellenden String. Die anderen beiden Parameter geben den Zeichensatz und die Farbe an, mit denen der **Label** abgebildet werden soll. **Labels** können einfacher mit **WidgetKit** erstellt werden.

1.3.4.2 MonoGlyph

Wie die Klasse **Glyph** ist auch diese von **Glyph** abgeleitete Klasse abstrakt. Ein **MonoGlyph** enthält genau einen **Glyph**, der als „*Body*“ bezeichnet wird, und verändert dessen Aussehen oder Verhalten in einer bestimmten Weise. Ein Objekt der **MonoGlyph**–Subklasse **Background** unterlegt beispielsweise einen **Glyph** mit einem farbigen Hintergrund.

MonoGlyph stellt Memberfunktionen zur Verfügung, die es erlauben, den *Body* abzufragen oder neu zu setzen. Die Semantik der Funktionen wird aus der Klassendeklaration deutlich:

```
class MonoGlyph : public Glyph {
  public:
    virtual void body(Glyph*);
    virtual Glyph* body() const;
    ...
};
```

Auf den enhaltenen **Glyph** kann mittels `MonoGlyph::body` zugegriffen werden. Wir werden dies noch später verwenden.

1.3.4.3 PolyGlyph

PolyGlyphs stellen die Möglichkeit dar, einen **Glyph** aus mehereren einzel–**Glyphs** zusammenzusetzen. Beispiele für **PolyGlyphs** sind Objekte des **LayoutKits**, z.B. die „hbox“, die ihre Komponenten von links nach rechts anordnet. Wir werden diese Objekte nicht selbst erzeugen, sondern zum Erstellen von **PolyGlyphs** die *Kits* **WidgetKit** und **LayoutKit** verwenden, die uns noch einiges an

Positionierungsarbeit der **Glyph**–Komponenten untereinander abnehmen. Trotzdem hier kurz die Klassendefinition:

```
class PolyGlyph : public Glyph {
  public:
    PolyGlyph(GlyphIndex initial_size = 10);
    virtual void append(Glyph*);
    virtual void prepend(Glyph*);
    ...
};
```

Durch `PolyGlyph::append` und `PolyGlyph::prepend` werden neue **Glyphs** an die vorhandenen entweder davor oder danach angehängt. Die Verhaltensweise dazu muß in abgeleitenden Klassen definiert werden.

1.3.4.4 Background

Diese Klasse ist von **MonoGlyph** abgeleitet. Sie ändert das Aussehen des enthaltenen **Glyph**–Objekts, indem sie unter dieses einen Hintergrund in einer bestimmten Farbe zeichnet. Wenn beispielsweise einem **ApplicationWindow** ein **Glyph**–Objekt übergeben wird, ist es sinnvoll, dieses in ein **Background**–Objekt zu *packen* und das so entstandene **Background**–Objekt an das **ApplicationWindow** zu übergeben. Die meisten **Glyphs** sind nämlich *durchsichtig*, d.h. der Fensterhintergrund ist durch den **Glyph** sichtbar. Dieser Fensterhintergrund wird aber oft von zufälligen Speichermustern gebildet, was zu unschönen Effekten führen kann.

Sehen wir uns den für diese Klasse interessanten Deklarationsausschnitt an:

```
class Background : public MonoGlyph {
  public:
    Background(Glyph* body, const Color*);
    ...
};
```

Als erstes Argument für den Konstruktor wird ein Zeiger auf den **Glyph** angegeben, unter den der Hintergrund gezeichnet werden soll. Schließlich wird noch ein Zeiger auf das **Color**–Objekt übergeben, das die Farbe des Hintergrunds bestimmt. Die Klasse **Color** soll an dieser Stelle noch nicht ausführlicher erklärt werden.

Abbildung 1.3 zeigt die genaue Ableitungsstruktur der Klassen **Label** und

Background. Diese Abbildung zeigt, daß die Klasse **Glyph** von **Resource** ab-

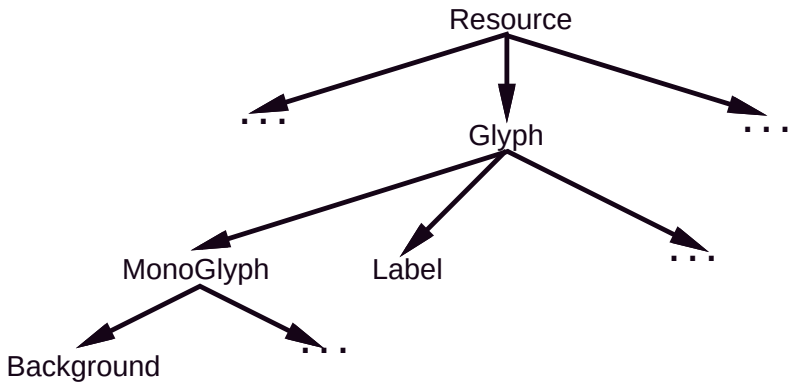


Abbildung 1.3: Einbettung von **Glyph**, Label und Background in die *InterViews* – Klassenhierarchie

geleitet ist. Damit sind z.B. alle Objekte der Klassen **Glyph**, **Background** und **Label** *shared objects*.

1.3.5 WidgetKit

Im Beispiel verwenden wir das *Kit* **WidgetKit**, und includen daher <**IV–look/kit.h**>. Dies ist eines der wichtigsten Tools, um Objekte für die Benutzeroberfläche zu generieren. **kit.h** wurde im Vergleich zu Version3.0 um einiges erweitert. Hier ein Auszug aus der Klassendefinition von **WidgetKit**:

```
class WidgetKit {
  public:
    static WidgetKit* instance();
    virtual Style* style() const;
    virtual const Color* foreground() const;
    virtual const Color* background() const;
    virtual Glyph* label(const char*) const;
    virtual Glyph* chiseled_label(const char*) const;
    virtual Glyph* raised_label(const char*) const;
    virtual Glyph* fancy_label(const char*) const;
    ...
    virtual Action* quit() const;
};
```

Die hier aufgeführten Bestandteile von **WidgetKit** stellen nur einen Bruchteil der wirklich zur Verfügung stehenden Funktionen dar. Dazu später mehr.

Der wichtigste Punkt bei Verwendung von **WidgetKit** ist die Art des Aufrufes. Jede *InterViews* – Anwendung hat maximal **ein WidgetKit**–Objekt. Deshalb wird dieses Objekt auch nicht mit „new“ erzeugt, sondern einer Adressvariablen mittels der `WidgetKit::instance` Memberfunktion zugewiesen. Falls zum Zeitpunkt des Aufrufs von `WidgetKit::instance` noch kein Objekt der Klasse **WidgetKit** erzeugt worden ist, wird dies hiermit automatisch erledigt. Damit steht das **WidgetKit** zur Verfügung und mit den Memberfunktionen können Standardwerte – Objekte für **Color**, **Font** und **Style** mit minimalem Aufwand erzeugt werden. Man beachte den Aufruf der Memberfunktionen mittels des „.“ – Operators.

1.3.6 Session

Der für dieses Beispiel interessante Deklarationsausschnitt der Klasse **Session** sieht folgendermaßen aus:

```
class Session {
  public:
    Session(const char*, int &argc, char **argv,
            const OptionDesc* = nil,
            const PropertyData* = nil);
    Style* style() const;
    void default_display(Display*);
    Display* default_display() const;
    virtual int run();
    virtual int run_window(Window*);
    virtual void quit();
    static Session* instance();
    ...
};
```

Jede *InterViews* – Anwendung muß als erstes genau ein **Session**–Objekt erzeugen. Das **Session**–Objekt übernimmt *Verwaltungsaufgaben* für eine *InterViews* – Sitzung. So erzeugt dieses Objekt beispielsweise automatisch ein **Display**–Objekt (wird als nächstes erklärt) für den Standardbildschirm. Das ist in der Regel derjenige, der in der Environmentvariablen **DISPLAY** enthalten ist.
Ein Zeiger auf dieses **Display**–Objekt wird von der Memberfunktion `Session::default_display` zurückgeliefert.

Wichtig ist auch die Memberfunktion `Session::run_window`. Sie bildet das als

Zeiger übergebene **Window**–Objekt (wird ebenfalls später erklärt) auf dem Bildschirm ab. Außerdem wird eine Schleife gestartet, die Eingabeereignisse wie z.B. das Drücken eines Mausknopfs abfrägt. *Für z.B. Animationen ist diese Funktion von Nachteil, da sie auf gewisse Eingabe–Ereignisse wartet und danach einzeln immer einen Refresh des Bildschirms ausführt. Falls Sie so etwas erstellen wollen, bietet sich die periodische Erzeugung von simulierten Ereignissen an (schwer), oder sie starten* `Session::run_window` *nicht und erledigen die enthaltenen Verwaltungsaufgaben in einer eigenen Funktion.*

Die Bedeutung der Parameter des Konstruktors der Klasse **Session** ist für eine exakte Beschreibung in diesem Tutorial zu umfangreich. Verwenden Sie in Ihren eigenen Programmen als ersten Parameter stets einen String, der irgendwie mit dem Programmnamen zusammenhängt, jedoch nicht unbedingt den Programmnamen selbst. Als zweites und drittes Argument geben Sie „**argc**“ und „**argv**“ aus der Kopfzeile der „**main**“–Funktion an. Die beiden letzten Parameter sollten Sie einfach weglassen und somit die Voreinstellung „nil“ akzeptieren. Die genaue Bedeutung dieser Parameter ebenso wie die der übrigen Memberfunktionen von **Session** können Sie als fortgeschrittener *InterViews* – Anwender einmal selbst im Referenzmanual [MAN3.1] nachschlagen.

1.3.7 Display

Der für uns wichtige Ausschnitt aus der Deklaration der Klasse **Display** hat folgende Gestalt:

```
class Display {
  public:
    virtual Coord width() const;
    virtual Coord height() const;
    virtual void style(Style*);
    virtual Style* style() const;
    virtual void repair();
    virtual void flush();
    virtual void sync();
    virtual boolean get(Event&);
    virtual void put(const Event&);
    ...
};
```

Ein Objekt dieser Klasse repräsentiert einen Bildschirm. Der Programmierer braucht ein solches Objekt nicht selbst zu erzeugen, da dies bereits von dem **Session**–Objekt einer *InterViews* – Applikation erledigt wird. Durch die Member-

funktion `Session::default_display` erhält er aber Zugriff auf diesen Standardbildschirm. Damit stehen ihm für dieses Objekt alle Memberfunktionen der Klasse **Display** zur Verfügung. Drei davon möchte ich an dieser Stelle kurz erwähnen:

Die Routinen `Display::width` und `Display::height` liefern die Breite und Höhe des zu dem **Display**–Objekt gehörenden Bildschirms. Die Memberfunktion `Display::style` gibt einen Zeiger auf das Standard – **Style**–Objekt (wird nachfolgend beschrieben) zurück, das automatisch mit dem **Display**–Objekt erzeugt wird.

1.3.8 Style

Auch für diese Klasse werden anhand eines Deklarationsausschnitts die anschaulichsten Memberfunktionen kurz erklärt:

```
  class Style : public Resource {
    public:
      Style();
      Style(const String& name);
      Style(const Style&);
      void name(const char*);
      void alias(const char*);
      void attribute(const char* name,
                     const char* value, int = 0);
      void remove_attribute(const char*);
      ...
};
```

Styles dienen dazu, gewisse Voreinstellungen zu treffen oder abzufragen, die die Ausgabe der *InterViews* – Anwendung am Bildschirm betreffen. Diese Klasse wurde gegenüber der Deklaration in Version 3.0 stark beschnitten; so findet hier nun nicht mehr die Anwahl von Zeigern auf Standardfarben für Vorder- und Hintergrund statt, ebensowenig wie für den Standardzeichensatz (Font).

Statt dessen wird jetzt **Style** nur noch für das Setzen von Attributen eines **Windows** verwendet, so zum Beispiel für den Namen in der Titelleiste mit `Style::attribute("name", char*)`. Mittels `Style::attribute` können auch noch andere vorgesehene Attribute gesetzt werden, schlagen Sie hierzu bitte bei Interesse im Referenzmanual [MAN3.1] nach.

Zu jedem **Display**–Objekt wird ein **Style**–Objekt erzeugt, das die Defaulteinstellungen für den entsprechenden Bildschirm enthält. **Styles** können aber auch unabhängig erzeugt werden. Bestimmte Objekte verlangen nämlich ein **Style**–Objekt bzw. einen Zeiger darauf als Parameter.

WidgetKit übernimmt viele Aufgaben des früheren **Style**–Objekts. Dazu später mehr.

1.3.9 Window/ApplicationWindow

Jede *InterViews* – Anwendung öffnet mindestens ein Fenster auf dem Bildschirm. Eines davon muß zum Hauptfenster bestimmt werden. Dieses Fenster wird als Objekt der Klasse **ApplicationWindow** erzeugt. **ApplicationWindow** ist von **ManagedWindow** abgeleitet, das wiederum von der Klasse **Window** abgeleitet ist (vgl. Abbildung 1.4). Das bedeutet, daß überall, wo ein Zeiger auf ein

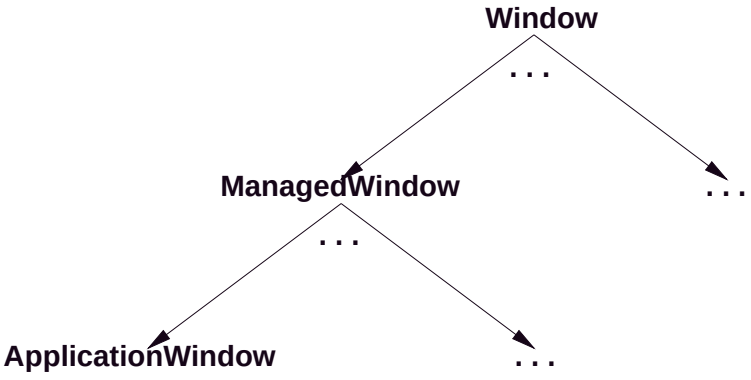


Abbildung 1.4: Ausschnitt aus der Windows – Klassenhierarchie

Window–Objekt als Argument verlangt ist, auch ein Zeiger auf ein **ApplicationWindow** stehen darf. Insbesondere ist ein Zeiger auf ein **ApplicationWindow** als Argument von `Session::run_window` erlaubt.

Pro Applikation sollte auch nur ein **ApplicationWindow** verwendet werden, bei weiteren darzustellenden **Windows** sollte man auf die anderen Möglichen **Windows** – Klassen zurückgreifen, um eine Hierarchie in der Fensterverwaltung zu bewahren.

Der interessante Deklarationsausschnitt der Klasse **ApplicationWindow** ist sehr kurz:

```
class ApplicationWindow : public ManagedWindow {
  public:
    ApplicationWindow(Glyph*);
    ...
};
```

Einem **Window** bzw. **ApplicationWindow** wird im Konstruktor ein Zeiger auf

ein **Glyph**–Objekt übergeben. Dieser **Glyph** wird automatisch in dem Fenster abgebildet.

Viel wichtiger sind jedoch die Funktionen, die von der Klasse **Window** an die Unterklasse **ApplicationWindow** vererbt werden:

```
class Window {
  public:
    virtual void style(Style*);
    Style* style() const;
    virtual void place(Coord left, Coord bottom);
    virtual void map();
    virtual void unmap();
    virtual boolean is_mapped() const;
    ...
}
```

Hier kann auch das **Style**–Objekt z.B. zur Namensgebung in der Titelleiste mittels `Window::style` übergeben werden. `Window::map` bildet das Fenster ab, und `Window::unmap` läßt es wieder vom Bildschirm verschwinden. `Window::place` plaziert das Fenster an einer Stelle auf dem Bildschirm.

1.4 Beispiel 1

Nachdem wir uns bisher ausschließlich mit theoretischen Fragen beschäftigt haben, soll jetzt ein erstes *InterViews* – Programm vorgestellt werden.

Das Programm bringt ein kleines Fenster mit fester Größe auf den Bildschirm, das den Text „Hallihallo, wie gehts denn so!“ enthält. Der zugehörige C++ – Code sieht folgendermaßen aus:

```
#include <IV-look/kit.h>
#include <InterViews/session.h>
#include <InterViews/window.h>
#include <InterViews/background.h>

int main(int argc, char **argv) {

   Session *session = new Session("Example",argc,argv);
   WidgetKit& kit = *WidgetKit::instance();

   return session->run_window(
          new ApplicationWindow(
                   new Background(
                       kit.label("Hallihallo, wie gehts denn so"),
                       kit.background()
                                 )
                          )
                        );
}
```

1.4.1 includes

Für alle Klassen, die verwendet werden, müssen natürlich vorher die zugehörigen Headerfiles eingebunden werden. Sie sollten auch für Ihre eigenen Klassen jeweils ein eigenes headerfile haben. Für die *InterViews* – Klassen gilt in der Regel, daß für jede Klasse, die explizit mit „new“ erzeugt wird, ihr Headerfile benötigt wird. Ansonsten sind oft einige Klassendeklarationen schon in übergreifenden Klassen, vorzugsweise *Kits*, enthalten und müssen daher nicht noch einmal extra angegeben werden. Um mehrfach – includes zu vermeiden, klammern Sie am besten die Headerfiles, wie in den Beispielen angegeben, mittels einer **#ifdefine** und **#endif** Konstruktion.

1.4.2 Beschreibung des Programmablaufs

Mit diesen – doch recht umfangreichen – Informationen sollten Sie jetzt – zumindest Ansatzweise – in der Lage sein, das Programm zu verstehen. In der ersten Zeile des Rumpfes von **main()** wird ein **Session**–Objekt erzeugt. In der zweiten Zeile wird eine Instance des **WidgetKits** generiert. In den folgenden Zeilen wird ein **ApplicationWindow** erzeugt. Durch `session-->run_window(*Window)` wird

dieses Fenster auf dem Bildschirm abgebildet und dabei wird eine Ereignisabfrageschleife gestartet. Die genaue Bedeutung dieser Schleife wird in Kapitel 3 beschrieben.

Das **ApplicationWindow** verlangt als Argument einen Zeiger auf ein Objekt der Klasse **Glyph**, oder einer der davon abgeleiteten Klassen. Dieses Objekt wird dann in dem Fenster dargestellt. Das verwendete **Background**–Objekt erfüllt diese Vorraussetzung. (siehe Abbildung 1.3). Der **Background** ist aber nur dazu da, sich wie ein Rahmen um seinen Inhalt zu legen, der aus den Zeigern auf ein **Glyph**– sowie ein **Color**–Objekt besteht. Diese generieren wir mittels `kit.label(char*)` und `kit.background()`, also als eigenlichem Inhalt ein Standard–**Label** mit Schriftzug „Hallihallo...“ in Standard– Vordergrundfarbe über der Standard– Hintergrundfarbe des Default–**Displays**.

Ich muß ehrlich gestehen, daß diese Flut an Informationen für den InterViews–Neuling bestimmt schwer zu verstehen ist. Daher empfehle ich Ihnen, zunächst das Programm wie im folgenden Unterabschnitt beschrieben zum Laufen zu bringen. Lesen Sie danach dieses Kapitel eventuell nochmals durch und beginnen Sie, mit dem Programm zu experimentieren. Vielleicht blättern Sie auch mal im Referenzmanual [MAN3.1] und sehen sich z.B. an, wie man eine Farbe mittels eines **Color**–Objekts erzeugt. Ersetzen Sie dann `kit.background()` durch eigene Farben oder lassen Sie den **Background** ganz weg, indem Sie direkt das **kit.label** – Objekt an das **ApplicationWindow** übergeben und gewinnen Sie so Sicherheit und Erfahrung im Umgang mit *InterViews*.

1.5 Compilieren von InterViews – Anwendungen

Wie in C bzw. C++ üblich, werden auch *InterViews* – Programme in der Regel mit Hilfe eines **Makefile** compiliert. Glücklicherweise bietet *InterViews* die Möglichkeit an, zugehörige **Makefiles** automatisch zu generieren, was auch absolut notwendig ist, wenn man die erforderlichen Pfad- und Variablendeklarationen für einen Übersetzungsvorgang betrachtet. Das verwendete Tool zum erzeugen des **Makefile** nennt sich „**ivmkmf**“, und sollte nach der Installation des *InterViews* – Pakets auf ihrem System im Standardpfad zugreifbar sein. Falls dies nicht der Fall ist, wenden Sie sich bitte an Ihren Systemadministrator oder suchen Sie in den *InterViews* – Systemdirectories nach dem Verzeichnis „**bin**“. Auch verlangt **ivmkmf**, daß Ihre Environment–Variable „**CPU**“ richtig gesetzt ist, um so den Pfad zu Ihren Systemspezifischen Daten eintragen zu können, da die *InterViews* – Applikation von vorneherein für die Verwaltung von mehreren verschiedenen Systemtypen gleichzeitig ausgelegt ist. (z.B. `setenv CPU='HP800'`).

Im folgenden ein Standard–Beispiel eines **Imakefile**, das die Grundlage für die Erzeugung des **Makefile** darstellt. Grundsätzlich wird für jede Applikation

ein eigenes **Makefile** und daher auch ein **Imakefile** benötigt.

```
SPECIAL_IMAKEFLAGS = -DUseInstalled -DTurnOptimizingOn=0

AOUT = hello

Use_libInterViews()

ComplexProgramTarget(a.out)
MakeObjectFromSrc(hello)
```

Zunächst werden zwei spezielle Makeflags, deren genaue Bedeutung hier nicht interessieren soll, eingestellt. Darauf folgend wird der Name festgelegt, den das ausführbare Programm anstatt von „**a.out**“ (Standardname) tragen soll. Verwenden Sie hier bitte **nicht** den Filenamen „**test**“, Sie werden sich sonst wundern, daß nichts passiert, wenn Sie das Programm ausführen wollen. (Zumindest wenn Sie an einem UNIX–System arbeiten).

Durch das Makro **Use_libInterViews()** wird Ihr Programm automatisch mit der *InterViews* – Standardbibliothek gebunden. Das Makro **ComplexProgramTarget** enthält den Filenamen des zu erzeugenden Files, hier ist es egal, ob Sie „**a.out**“ oder auch den Filenamen eintragen, den das Executable tragen soll.

Bleibt noch, die Quelldatei(en) anzugeben, in der/denen der zu übersetzenden Programmtext enthalten ist. Hier benötigen Sie für jede einzelne Datei einen Aufruf von **MakeObjectFromSrc(***Dateiname***)**, wobei *Dateiname* der jeweilige Name der Quelldatei ist, ohne die verwandte Standard–C Endung „**.c**“ („**hello.c**“ übersetzen Sie also wie oben angegeben zu dem ausführbaren File „**hello**“.)

Geben Sie jetzt bitte das *InterViews* – Beispielprogramm von Seite 16 unter dem Namen „**hello.c**“ und obiges **Imakefile** ein. Erzeugen Sie mittels „**ivmkmf**“ ein **Makefile**, und starten Sie dieses anschließend mit „**make**“. Nach erfolgreichem Übersetzungsvorgang müßten Sie jetzt ein File „**hello**“ in Ihrem Verzeichnis vorfinden. Starten Sie dieses und erfreuen Sie sich an Ihrer ersten *InterViews* – Applikation. Sie können das Programm bisher nur durch Zerstören des Prozesses von außen beenden, indem Sie z.B. <CTRL>–C in dem Fenster betätigen, von dem aus Sie die Applikation gestartet haben. Das liegt daran, daß Sie einen sog. „**main event–loop**“ (Endlosschleife zur Abfrage von Eingabeereignissen) mittels **session–>run_window(window*)** gestartet haben, der in unserem Beispiel nirgends verlassen werden kann. Die reguläre Beendigung einer *InterViews* – Applikation wird erst in Kapitel 3 behandelt.

Kapitel 2

Ausgabe auf den Bildschirm

In diesem Kapitel wollen wir uns der Darstellung von Graphiken auf dem Bildschirm zuwenden, die aus einfachen Bestandteilen wie z.B. Linien oder Polygonen bestehen. Zum besseren Verständnis werden wir diese anhand eines kleinen Beispielprogramms realisieren.

Die Besprechung der dazu notwendigen *InterViews* – Klassen habe ich hier wieder an den Anfang gestellt; Sie können jedoch genauso zuerst das am Ende des Kapitels vorgestellte Programm eingeben, und dann anhand diesem mit den einzelnen Klassen experimentieren und so die Definitionen lernen.

Den Programmtext mit zugehörigem **Imakefile** finden Sie in Abschnitt 2.3 auf der Seite 33.

2.1 Aufgabenstellung

Mir kommt es in dem zweiten Kapitel dieses Tutorials nur darauf an, Ihnen prinzipiell zu erklären, wie in *InterViews* Bildschirmausgaben erzeugt werden. Deshalb wollen wir hier exemplarisch nur eine relativ einfache Aufgabe lösen. Auf dem Bildschirm soll ein Fenster erscheinen, das eine schwarze Linie von links unten nach rechts oben enthält. Zusätzlich soll dieses Fenster vom Benutzer in bestimmten Schranken vergrößert und verkleinert werden können. Unabhängig von der Fenstergröße soll dabei die Linie aber stets automatisch vom linken unteren zum rechten oberen Fensterrand gezeichnet werden. Abbildung 2.1 zeigt einen Screendump der Lösung. Das Aussehen des Fensterrahmens ist dabei abhängig von dem verwendeten Windowmanager und kann demzufolge auf Ihrem System anders gestaltet sein. Wichtig ist der Inhalt des Fensters, der von unserem *InterViews* – Programm erzeugt werden soll.

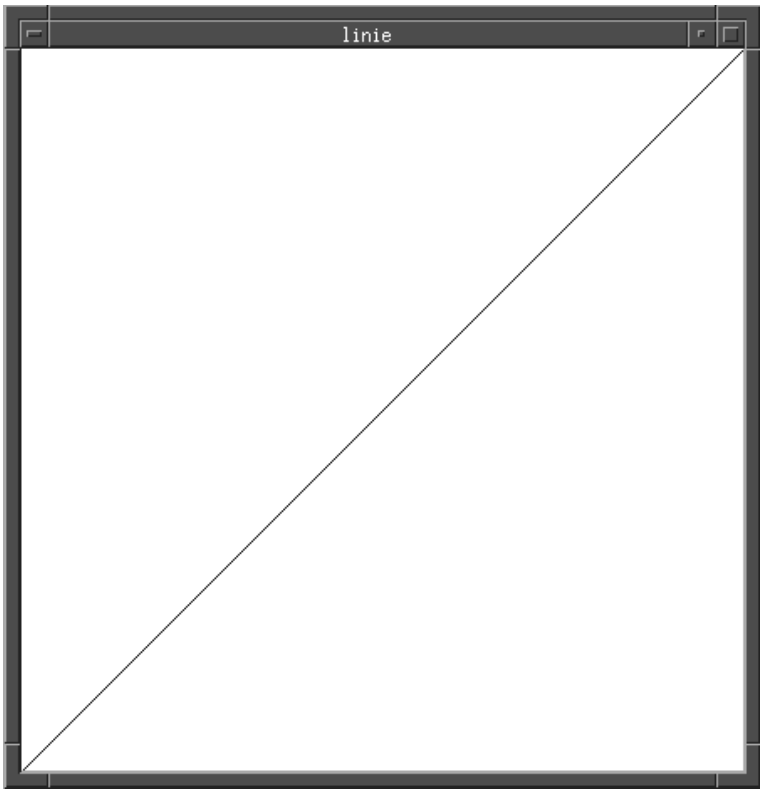


Abbildung 2.1: Bildschirmausgabe des Beispielprogramms von Kapitel 2

2.2 Notwendige InterViews – Klassen

Im folgenden werden die *InterViews* – Klassen vorgestellt, die zusätzlich zu denen im vorhergenhenden Kapitel zur Lösung der Aufgabe nötig sind.

Außerdem werde ich einige schon besprochene Dinge nochmals wiederholen und manchmal genauer auf eine Klasse eingehen, als dies bisher geschehen ist. Dies soll Ihnen helfen, sich wichtige Sachverhalte besser einprägen zu können.

2.2.1 Color

Objekte der Klasse **Color** dienen der Erzeugung von Farben. Der für uns wichtige Deklarationsausschnitt hat folgende Gestalt:

```
typedef float ColorIntensity;
typedef unsigned int ColorOp;

class Color : public Resource {
  public:
    enum { Copy, Xor, Invisible };
    Color(ColorIntensity red, ColorIntensity green,
          ColorIntensity blue,
          float alpha = 1.0, ColorOp = Copy
    );
    ...
};
```

Die Klasse **Color** ist von **Resource** abgeleitet, d.h. Objekte dieser Klasse gehören zu den *shared objects* und dürfen nicht mit *delete* gelöscht werden!

Von den Memberfunktionen ist im Moment nur der Konstruktor wichtig, die übrigen können Sie selbst im Referenzmanual [MAN3.1] nachschlagen. Die ersten drei Argumente geben den Rot-, Grün- und Blauanteil der zu erzeugenden Farbe an. Sie können jeweils beliebige Floatwerte zwischen **0.0f** und **1.0f** angeben. **1.0f** steht für die volle Intensität des entsprechenden Farbanteils. **0.0f** zeigt dagegen an, daß der betreffende Farbanteil in der erzeugten Farbe fehlt. *(Vergessen Sie nicht, daß das „***f***“ hinter der Zahl bedeutet, daß es sich um einen „***float***“ – Wert handelt. Die Konstante alleine würde dagegen einen „***double***“ – Wert darstellen, was hier bei guten Compilern zu Fehlermeldungen führt.)*

Die beiden letzten Argumente werden normalerweise nicht angegeben, weil die eingestellten Defaultwerte den Effekt erzeugen, den man intuitiv erwartet. Mit dem vierten Parameter könnten Sie die *Sichtbarkeit* der Farbe regeln. Dabei sind wieder Werte zwischen **0.0f** und **1.0f** möglich, z.B. bedeutet **alpha = 1.0f**, daß die Farbe *voll sichtbar* ist. Der Wert **Copy** für den letzten Parameter bewirkt, daß beispielsweise mit Linien einer so erzeugten Farbe der Hintergrund übermalt wird. Wenn Sie über einen Monochrombildschirm verfügen und **Xor** angeben, wird die Linie mit dem Hintergrund exklusiv–oder–verknüpft. Dies bedeutet, daß bereits dunkle Bildschirmpunkte wieder hell werden, wenn Sie darüber eine dunkle Linie zeichnen.

Wenn Sie eine bestimmte Farbe erzeugen wollen, müssen Sie bei der Angabe der Rot-, Grün- und Blauanteile darauf achten, daß wir es hier mit additiver

Farbmischung zu tun haben. Zur Erleichterung sind einige Argumenttripel mit den zugehörigen Farben angegeben:

```
new Color(0.0f,0.0f,0.0f);     Schwarz;
new Color(1.0f,1.0f,1.0f);     Weiss;
new Color(1.0f,0.0f,0.0f);     helles Rot;
new Color(0.0f,0.0f,0.5f);     dunkles Blau;
new Color(0.8f,0.0f,0.8f);     Lila.
```

Eine sehr schöne Farbtafel, die die Zusammensetzung vieler Mischfarben übersichtlich darstellt, ist in [PI] auf Seite 76 zu finden.

Bitte beachten Sie, daß aufgrund eines *InterViews* – Fehlers innnerhalb einer Applikation nicht mehr als ca. 250 Farben korrekt erzeugt werden können. Die Erzeugung weiterer Farben ist zwar möglich, doch entsprechen diese nicht mehr der Farbspezifikation, die Sie im Konstruktor angegeben haben. (*Zumindest war dies Stand der Dinge im Tutorial 3.0, ob dies in Version 3.1 immer noch zutrifft, habe ich nicht geprüft. Anm. d. A.*)

2.2.2 Brush

Objekte der Klasse **Brush** werden vielen Zeichenoperationen übergeben, um Strichstärke und Strichmuster zu bestimmen. (englisch „Brush“ entspricht dem deutschen „Pinsel“). Anhand eines Deklarationsausschnitts soll diese Klasse kurz erläutert werden:

```
class Brush : public Resource {
  public:
    Brush(Coord width);
    Brush(int pattern, Coord width);
    ...
};
```

Wie **Color** ist auch **Brush** von **Resource** abgeleitet. Demzufolge gehören Objekte der Klasse **Brush** ebenfalls zu den *shared objects*.

Für unsere Zwecke reicht der oben aufgeführte erste Konstruktor aus. Als Parameter wird ihm die gewünschte Strichstärke des zu erzeugenden Pinsels übergeben. Mit dem zweiten Konstruktor kann auch noch das Strichmuster spezifiziert werden. Sehen Sie dazu bitte bei Interesse im Referenzmanual [MAN3.1] nach.

2.2.3 Canvas

Ein **Canvas**–Objekt repräsentiert eine *Leinwand*, auf der gezeichnet werden kann. Die von *InterViews* bereitgestellten Fensterobjekte, z.B. **ApplicationWindow**, erzeugen automatisch ein **Canvas**–Objekt. Die Spezifizierung, was auf der *Leinwand* gezeichnet werden soll, erfolgt in der `Glyph::draw` – Funktion des **Glyphs**, der dem Fenster im Konstruktor übergeben wird. Der vorgestellte Deklarationsausschnitt fällt für die Klasse **Canvas** etwas umfangreicher aus:

```
class Canvas {
  public:
    virtual Window* window() const;
    virtual Coord width() const;
    virtual Coord height() const;
    virtual void new_path();
    virtual void move_to(Coord x, Coord y);
    virtual void line_to(Coord x, Coord y);
    virtual void curve_to(Coord x, Coord y, Coord x1,
                          Coord y1, Coord x2, Coord y2);
    virtual void close_path();
    virtual void stroke(const Color*, const Brush*);
    virtual void fill(const Color*);
    virtual void line(Coord x1, Coord y1, Coord x2,
                      Coord y2, const Color*, const Brush*);
    virtual void rect(Coord l, Coord b, Coord r, Coord t,
                      const Color*, const Brush*);
    virtual void fill_rect(Coord l, Coord b, Coord r,
                           Coord t, const Color*);
    virtual void character(const Font*, long ch, Coord width,
                           const Color*, Coord x, Coord y);
    virtual void damage_all();
    virtual void redraw(Coord left, Coord bottom,
                        Coord right, Coord top);
};
```

Da die Objekte der Klassen, die *InterViews* für Bildschirmfenster bereitstellt (**Window** und davon abgeleitete Klassen), jeweils selbst einen **Canvas** erzeugen (siehe oben), braucht der *InterViews* – Programmierer dies im Regelfall nicht selbst zu tun.

Die Memberfunktion `Canvas::window` liefert einen Zeiger auf das Fenster, zu dem das **Canvas**–Objekt gehört, `Canvas::width` und `Canvas::height` liefern

die Ausdehnung der *Leinwand* in Breite und Höhe. Die übrigen oben angeführten Memberroutinen dienen dazu, direkt auf den **Canvas** zu zeichnen. Die Zeichenoperationen richten sich dabei nach dem *Adobe* – Standard für Postscript Seitenbeschreibungen. Wer damit schon zu tun hatte, wird sich mit diesen Funktionen wahrscheinlich auskennen. Für alle anderen sollen sie im nachfolgenden beschrieben werden. Ziel dieser Art der Zeichenoperationen ist es, Ausgaben sowohl auf dem Bildschirm als auch auf einem Drucker erzeugen zu können. Die `Glyph::draw` – Routine bleibt in diesem Fall dieselbe und wird entweder einem **Canvas** oder einem **Printer**–Objekt übergeben. Wir wollen hier jedoch nicht weiter darauf eingehen; bei Interesse lesen Sie bitte Kapitel 6 im Referenzmanual [MAN3.1].

Jetzt wollen wir uns also ansehen, wie mit den einfachen Memberroutinen der Klasse **Canvas** Zeichnungen erzeugt werden können.
Mittels `Canvas::new_path`, `Canvas::move_to`, `Canvas::line_to`, `Canvas::curve_to`, und `Canvas::close_path` wird über eine Liste von Coordinaten das Gebilde festgelegt, das gezeichnet werden soll:

- mit new_path() wird ein neuer *Pfad* geöffnet.
- move_to() springt zu einem Koordinatenpunkt, ohne eine Linie dorthin zu ziehen.
- line_to und curve_to verbinden den bisherigen und den neu angegebenen Koordinatenpunkt durch eine Linie bzw. durch eine *Bezier*–Kurve.
- Mittels close_path() wird der *Pfad* geschlossen.

Der so angegebene *Pfad* kann auf zwei Arten am Bildschirm abgebildet werden:

- Durch `Canvas::stroke` wird er mit der übergebenen Farbe und dem übergebenen Pinsel als Linienzug gezeichnet.
- Der Aufruf von `Canvas::fill` bewirkt, daß die von dem Pfad eingeschlossene Fläche mit der angegebenen Farbe ausgefüllt wird.

Um eine schwarz umrandete, grau gefüllte Fläche zu erzeugen, kann auch die Kombination beider Aufrufe sinnvoll sein. Als Beispiel sehen wir uns folgende Pfaddefinition an:

```
canvas->new_path();       // canvas ist vom Typ Canvas*
canvas->move_to(x1,y1);   // x1, y1 vom Typ Coord
canvas->line_to(x2,y2);   // x2, y2 vom Typ Coord
canvas->line_to(x3,y3);   // x3, y3 vom Typ Coord
canvas->close_path();

// canvas->stroke(Color*,Brush*) oder canvas->fill(Color*);
```

Im Anschluß an diese Pfaddefinition können Sie `Canvas::stroke` oder `Canvas::fill` aufrufen. Das Ergebnis dieser beiden Routinen ist in Abbildung 2.2 zu sehen.
`Canvas::line` ist das Äquivalent zu einem Pfad mit zwei Punkten, hierzu muß

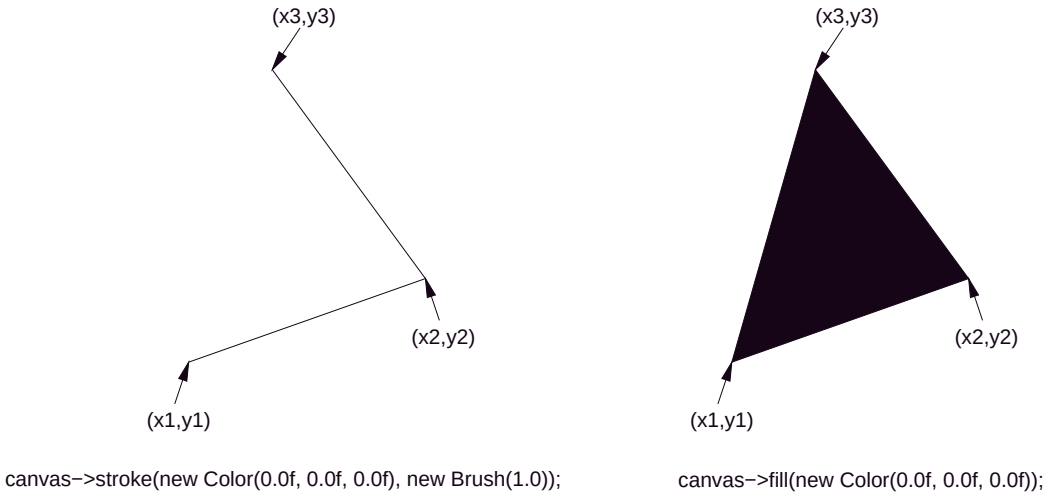


Abbildung 2.2: Auswirkung der Routinen stroke und fill auf einen zuvor spezifizierten Pfad

jedoch nicht erst ein Pfad geöffnet und geschlossen werden; auch der Befehl `Canvas::stroke` ist hier schon enthalten. Genauso verhalten sich die Funktionen `Canvas::rect`, `Canvas::fill_rect` und `Canvas::character`. Hierbei wird ein Rechteck, ein mit der angegebenen Farbe ausgefülltes Rechteck sowie ein Buchstabe in angegebenen Font auf den spezifizierten Koordinaten dargestellt. `Canvas::damage_all` und `Canvas::redraw` veranlassen ein Neuzeichnen des **Canvas** in festgelegten Schranken. Besser ist es jedoch, den **Glyph**, der die „draw“ – Funktion für den **Canvas** enthält, einem **InputHandler** zu übergeben, der dann wiederum mit `InputHandler::redraw` den **Canvas** updated; dazu aber später.

2.2.4 Glyph

Objekte dieser Klasse sind die elementaren Bausteine für graphische Oberflächen unter *InterViews*, wie bereits in Kapitel 1 angedeutet wurde. Ebenfalls in Kapitel 1 wurde erklärt, daß einem **Window**–Objekt ein Zeiger auf einen **Glyph** übergeben wird. Wird das Fenster auf dem Bildschirm abgebildet, so wird automatisch dieser **Glyph** auf dem **Canvas** des **Windows** abgebildet.

Wir wollen uns nun anhand eines Deklarationsausschnitts die elementaren Memberfunktionen der Klasse ansehen:

```
class Glyph : Resource {
  protected:
    Glyph();
  public:
    virtual void request(Requisition&) const;
    virtual void allocate(Canvas*,const Allocation&,
                          Extension&);
    virtual void draw(Canvas*,const Allocation&) const;
    ...
};
```

Der Konstruktor der Klasse ist geschützt, d.h. er steht nur abgeleiteten Klassen zur Verfügung. Damit können Sie selbst kein Objekt der Klasse **Glyph** erzeugen. Dies ist auch sinnvoll, da diese Basisklasse nur ein *Protokoll* für Objekte anbieten soll, die in einem Bildschirmfenster abgebildet werden. Die oben angegebenen virtuellen Memberfunktionen regeln die Kommunikation zwischen **Glyphs** und dem System. Das konkrete Verhalten dieser Funktionen müssen Sie selbst implementieren, indem Sie von **Glyph** eigene Klassen ableiten und dort die virtuellen Funktionen `Glyph::request`, `Glyph::allocate` und `Glyph::draw` neu ausprogrammieren. *Die grundsätzliche Vorgehensweise von InterViews – Programmierern besteht darin, von solchen „Gerüstklassen“ eigene Klassen abzuleiten. Durch Implementation bzw. Reimplementation der Funktionen und Hinzufügen neuer Memberfunktionen können sie die abgeleiteten Klassen ihren Vorstellungen entsprechend gestalten. InterViews bietet zu Ihrer Arbeitserleichterung bereits viele solcher abgeleiteter Klassen an, z.B. die von* **Glyph** *abgeleitete Klasse* **Label**, *die Sie bereits im Beispiel aus Kapitel 1 kennengelernt haben.*

Im folgenden wird nun das **Glyph**–Protokoll anhand der drei wichtigsten Memberfunktionen „**request**“, „**allocate**“ und „**draw**“ besprochen.

Wenn ein Fenster auf dem Bildschirm abgebildet werden soll, versucht das System, den **Glyph** zu zeichen, der dem **Window**–Objekt im Konstruktor übergeben wurde. Dabei laufen im einzelnen folgende Schritte ab:

- Zunächst ruft das Laufzeitsystem die Memberfunktion `Glyph::request` auf. In **request** muß das **Glyph** – Programm in dem übergebenen Parameter **Requisition** angeben, wie und welchen Koordinatenbereich der **Glyph** auf dem Bildschirm einnehmen soll. Aus diesen Daten wird dann im Einzelfall berechnet, welche Größe dem **Glyph** zugewiesen werden kann.

- Danach ruft das System `Glyph::allocate` auf. In dieser Funktion wird dem **Canvas** der **Glyph** mitgeteilt, den er darstellen soll. In dem Parameter **Allocation** wurde inzwischen vom System der Koordinatenbereich berechnet, der dem **Glyph** aufgrund seiner Anforderung in **request** tatsächlich zugeteilt wird. Dieser Bereich kann von der Anforderung abweichen, vor allem, wenn das Fenster vom Windowmanager (z.B. durch Mausziehen) verändert wurde.

- Zum Abschluß wird `Glyph::draw` mit denselben Parametern **Canvas**“ und **Allocation** wie zuvor **allocate** aufgerufen. In dieser Routine sollen die Zeichenoperationen des **Glyphs** implementiert werden.

Der Programmierer ruft im Regelfall keine dieser drei Memberfunktionen selbst auf. So hat beispielsweise ein expliziter Aufruf von `Glyph::draw` keinen Effekt, wenn Sie nicht die zugehörigen anderen Funktionen aufrufen, um dies auch darzustellen. Deswegen bedienen Sie sich hierbei besser der Funktion `Canvas::redraw` oder `InputHandler::redraw`, mit letzterer habe ich meine besten Erfahrungen gemacht.

Die Headerfiles zu den im folgenden beschriebenen Klassen **Requirement**, **Requisition**, **Allotment**, **Extension**, *etc. finden sich in* **geometry.h**, *und nicht in eigenen Headerfiles für jede Klasse, wie sonst üblich. Dies nur als Anmerkung, um Ihnen die eventuelle Suche zu ersparen.*

2.2.5 Requisition

Die Memberfunktion `Glyph::request` verlangt die Übergabe einer **Requisition**, in der die Anforderungen an das Fenster für jede verwendete Dimension festgelegt werden sollen. Welche Anforderungen nun genau gestellt werden, wird in der Klasse **Requirement** festgelegt. **Requisition** vereint nur die **Requirements** verschiedener Dimensionen auf ein Objekt.

Die Klasse **Requisition** ist folgendermaßen definiert:

```
typedef unsigned int DimensionName;

enum {
  Dimension_X = 0, Dimension_Y, Dimension_Z, Dimension_Undefined
};

class Requisition {
  public:
    void require(DimensionName, const Requirement&);
    Requirement& requirement(DimensionName);
    ...
};
```

Ein **Requisition**–Objekt enthält für jede Dimension genau ein **Requirement**–Objekt. Da in unserem Beispiel ein **Glyph** auf eine zweidimensionale *Leinwand* (**Canvas**) gezeichnet werden soll, werden wir je ein **Requirement** für die X- und Y-Dimension mittels der Memberfunktion `Requisition::require` definieren. Die Memberfunktion `Requisition::requirement` gibt wieder das aus, was vorher in **require** festgelegt wurde.

2.2.6 Requirement

Die eigentliche Anforderung, die an das Fenster gestellt wird, wird für jede Dimension in einem eigenen **Requirement** festgelegt. Es handelt sich hierbei um Geforderte Größen für *natürliche Größe*, *Dehnbarkeit*, *Stauchbarkeit* und *Alignment*.

```
class Requirement {
  public:
    Requirement(Coord natural, Coord stretch,
                Coord shrink, float alignment);
    Coord natural() const;
    Coord stretch() const;
    Coord shrink() const;
    float alignment() const;
    ...
};
```

Es sind auch noch drei andere Konstruktoren für die Klasse **Requirement**

möglich, auf diese soll jedoch hier nicht eingegangen werden.

Der erste Parameter gibt die *natürliche Größe* eines Objektes an. Dies ist die normale Ausdehnung des **Glyphs**, wenn er nicht gedehnt oder gestaucht wird (was bei großen **Glyphs** schon zu Beginn der Darstellung passieren kann, wenn dieser eventuell größer als der Bildschirm ist !). Wenn Ihr Windowmanager es gestattet, das Fenster, in dem der **Glyph** abgebildet ist, in gewissen Schranken zu verkleinern oder zu vergrößern, dann werden genau diese Schranken durch die beiden Parameter **stretch** und **shrink** festgelegt. In dem Maße, wie das Fenster vergrößert wird, wird dann auch der **Glyph** vergrößert, und wenn die „**draw**" – Funktion hierfür Spielraum lässt, auch die Zeichnung darin. Hierbei ist die *maximale Größe* die *natürliche Größe* (**natural**) plus die *Dehnbarkeit* (**stretch**), die *minimale Größe* ist die *natürliche Größe* minus der *Stauchbarkeit* (**shrink**).

Mit dem Parameter „**align**" können Sie für jede Dimension einzeln die Lage des Koordinatenursprungs für den zu zeichnenden **Glyph** festlegen. Diesen können Sie also frei auf dem **Canvas** bewegen. Für „**align**" sind Werte zwischen **0.0f** und **1.0f** erlaubt. **0.0f** bedeutet, daß der Koordinatenursprung in der betreffenden Dimension bei der kleinsten Koordinate des Objekts liegt. Entsprechend setzt der Wert **1.0f** den Ursprung bei der größten Koordinate fest. *In InterViews wachsen Koordinaten in x–Richtung von links nach rechts und in y–Richtung von unten nach oben.*

Im folgenden sind einige Paare für **alignment**–Werte (x,y) mit dem dazugehörigen Koordinatenursprung angegeben. x gibt den **align**–Wert des **Requirement**–Objekts für die x–Richtung an. Entsprechend enthält y den **align**–Wert für das **Requirement**–Objekt in y–Richtung:

x-Alignment	y-Alignment	Position des Ursprungs
0.0f	0.0f	links unten
0.5f	0.0f	unten mitte
1.0f	0.0f	unten rechts
0.0f	0.5f	links mitte
0.5f	0.5f	Zentrum
1.0f	1.0f	rechts oben

In unserem Beispiel legen wir den Ursprung des **Linie**–Objekts durch die Angabe von **align = 0.0f** für beide Dimensionen in die linke untere Ecke.

Obwohl der Rumpf von `Glyph::allocate` nur eine Zeile umfaßt, wird die Erklärung dieser Memberfunktion ebenfalls etwas breiteren Raum einnehmen, weil ich an dieser Stelle die Klassen **Allocation**, **Allotment** und **Extension** erklären möchte.

2.2.7 Allocation

Ebenso wie `Glyph::request` wird `Glyph::allocate` vom System aufgerufen. Die Memberfunktion **allocate** wird nicht nur dann aufgerufen, wenn das **Glyph**–Objekt zum erstenmal auf dem Bildschirm erscheint, sondern jedesmal, wenn sich etwas ändert, z.B. wenn das zugehörige Fenster vergrößert wird. Dabei teilt die übergebene **Allocation** dem Glyph mit, welche Größe er jetzt aktuell annehmen soll.

Im folgenden der interessantere Teil des Headerfiles der Klasse **Allocation**:

```
class Allocation {
  public:
    Allocation();
    void allot(DimensionName, const Allotment&);
    Allotment& allotment(DimensionName);
    Coord left() const;
    Coord right() const;
    Coord bottom() const;
    Coord top() const;
    ...
};
```

Ebenso wie ein **Requisition**–Objekt enthält auch ein **Allocation**–Objekt für jede einzelne Dimension je ein Objekt, hier ist dies ein Objekt der Klasse **Allotment**. Die Klasse **Allocation** vereint wieder nur die **Allotments** verschiedener Dimensionen auf ein Objekt.

Die **Allotment**–Objekte für die jeweiligen Dimensionen werden unter Angabe des Dimensionsnamens mittels `Allocation::allot` bzw.
`Allocation::allotment` gesetzt bzw. ausgelesen.

Die Memberfunktionen der Klasse **Allocation**, die bisher noch nicht erklärt wurden, erleichtern die lesenden Zugriffe auf die in den einzelnen **Allotments** gespeicherten Daten. Die Namen der Routinen sind selbsterklärend, daher wird hier nur ein kleines Beispiel zum Verständnis vorgeführt:

Sei z.B. **alloc** ein **Allocation**–Objekt. Dann ist **alloc.left()** gleichbedeutend mit **alloc.allotment(Dimension_X).begin()**.

2.2.8 Allotment

Ein **Allotment** spezifiziert die aktuell errechnete Größe des **Glyphs** für eine Dimension. Die Größe wird durch drei Werte angegeben: **Origin**, **span** und **alignment**. **Origin** ist eine Position in dem **Allotment** und **span** die Größe des

Allotments. **Alignment** ist eine Zahl zwischen **0.0f** und **1.0f**, die den Koordinatenursprung festlegt.

Im folgenden der Interessante Teil des Header–Files der Klasse **Allotment**:

```
class Allotment {
  public:
    Allotment(Coord origin, Coord span, float alignment);
    Coord origin() const;
    Coord span() const;
    float alignment() const;
    Coord begin() const;
    Coord end() const;
    ...
};
```

Die Memberfunktionen erlauben den Zugriff auf die charakteristischen Größen eines **Allotment**–Objekts; `Allotment::begin` liefert die kleinste Koordinate des Koordinatenbereichs, der durch das **Allotment**–Objekt repräsentiert wird. Analog liefert `Allotment::end` dessen größte Koordinate `Allotment::span` gibt die Ausdehnung an, die der Glyph animmt. `Alotment::origin` liefert die absolute und `Allotment::alignment` die relative Position des Koordinatenursprungs. Für ein **Allotment**–Objekt „**a**“ könnten beispielsweise folgende Werte errechnet werden:

```
a.span() = 100.0;
a.origin() = 20.0;
a.alignment() = 0.4;
a.begin() = -20.0;
a.end() = 80.0.
```

Ein Vergleich mit den Angeforderten Größen von **Requirement** liefert: (Zur Erinnerung: ein **Requirement** besteht aus den angeforderten Werten „**natural**“, „**stretch**“, „**shrink**“ und „**alignment**“.) Der Wert von „**span**“ ist die berechnete Ausdehnung des **Glyphs**, er kann zwischen „**natural**“ + „**stretch**“ und „**natural**“ − „**shrink**“ liegen (also zwischen angefordertem Minimum und Maximum); „**alignment**“ und „**origin**“ ergeben sich aus dem angeforderten **alignment** und der aktuellen Position des **Glyphs** auf dem **Canvas** des **Windows**; die Koordinaten reichen von −20 (**begin**) bis **80** (**end**) in dieser Dimension, der Koordinatenursprung liegt dann beim Wert **20.0**, also bei relativ **0.4** (von **0.0** bis **1.0**).

Man beachte hierbei, daß der **Glyph** *nicht unbedingt den* **Canvas** *ausfüllen*

muß, und deswegen der Koordinatenursprung etwas weiter zur Mitte liegen kann, selbst wenn man ihn außen am Rand gefordert hat. Um dies zu umgehen, kann man direkt die Ausdehnung des **Canvas** *in die „***draw***“ – Funktion des* **Glyphs** *einbeziehen. Wir werden dies in unserem Beispiel anwenden.*

2.2.9 Extension

Für das Verständnis der Parameterleiste der Funktion `Linie::allocate` fehlt nur noch eine Erklärung des Parameters vom Typ **Extension**. Dieser soll jetzt kurz beschrieben werden.

Das **Extension**–Objekt ist das Objekt, das dem **Canvas** und damit dem **Display** letztendlich die genaue Größe und Ausdehnung des **Glyphs** übergibt und damit festlegt. Meist wird der **Extension** nur das eben in **Allocation** berechnete Ergebnis übergeben. Die Zeile „**ext.set(c, a)**“ in `Linie::allocate` erfüllt diese Aufgabe. Diese Zeile sollte in jeder Ihrer selbstgeschriebenen **allocate** – Routinen ganz am Anfang stehen. Es wäre aber auch möglich, hier eigene Werte einzugeben. Bei Interesse und Neugier können Sie das ja gerne einmal ausprobieren.

```
class Extension {
  public:
    void set(Canvas*, const Allocation&);
    void set_xy(Canvas*, Coord left, Coord bottom,
                         Coord right, Coord top);
    void clear();
    void merge(Canvas*, const Allocation&);
    void merge_xy(Canvas*, Coord left, Coord bottom,
                           Coord right, Coord top);
    ...
};
```

Wenn Sie Ihren **Glyph** komplett neu zeichnen wollen, müssen Sie die **Extension** dieses **Glyphs** neu laden. `Extension::set` initialisiert eine **Extension** auf die gegebenen **Allocation**–Parameter, gerundet, um auf den **Canvas** zu passen. `Extension::clear` setzt die **Extension** auf eine leere Umgebung.

`Extension::merge` erweitert eine **Extension**, so daß sie eine neue Umgebung zusätzlich zu ihrer alten enthält. `Extension::merge_xy` regelt die Größe einer **Extension**, sodaß sie eine Rechteck enthält, das durch die Grenzen „**left**“, „**bottom**“, „**right**“ und „**top**“ angegeben wird.

2.3 Beispiel 2

Die eigentliche Implementierung unserer Aufgabe ist wesentlich weniger umfangreich, als man es nach dieser doch recht ausführlichen Klassenbeschreibung vermuten könnte. Der Grund dafür ist, daß *InterViews* wirklich sehr einfach zu programmieren ist, *wenn man weiß, wie es funktioniert.*
Und um das zu lernen, läßt sich ein gewisser Aufwand leider nicht vermeiden. Aber Sie werden sehen, daß es sich lohnt!

Die Lösung unserer Aufgabe ist in die vier Files **header.h**, **linie.h**, **linie.c** und **main.c** unterteilt. Die Datei **header.h** enthält eine Zusammenfassung der benötigten Headerfiles:

```
#ifndef header_h
#define header_h

#include <IV-look/kit.h>

#include <InterViews/session.h>
#include <InterViews/window.h>
#include <InterViews/background.h>
#include <InterViews/color.h>
#include <InterViews/brush.h>
#include <stream.h>

#include "linie.h"

#endif
```

Die Datei „**stream.h**“ wurde hier aufgenommen, um Ihnen in der Testphase des Programms die Ausgabe von **cerr**–Meldungen auf den Bildschirm zu ermöglichen.

Wie schon in Kapitel 1 besprochen, verwende ich für jede Klasse ein eigenes „**.c**“ – File und ein eigenes Header – File. Sie sollten sich auch daran gewöhnen. Selbst bei kleineren Programmen können sie schon bald wieder Klassen verwenden, die Sie vorher schon programmiert haben, und sie sind dann froh, wenn Sie ihre Files nicht erst wieder zerpflücken müssen. Gleichzeitig empfiehlt sich die Verwendung eines gemeinsamen **header.h** – Files, um zumindest die **#includes** der *InterViews* – Umgebung nur einmal einzubinden. Wo sie die Headerdateien ihrer eigenen Klassen dann einbinden, ob in den jeweiligen „**.c**“ – Files oder auch in **header.h**, bleibt Ihnen überlassen. Ich empfehle jedoch aus Erfahrung meine Version.

Die Datei **linie.h** enthält die Deklaration der von **Glyph** abgeleiteten Klasse **Linie**:

```
#ifndef linie_h
#define linie_h

class Linie : public Glyph {
public:
        Linie();
        virtual ~Linie();
        virtual void request(Requisition&) const;
        virtual void allocate(Canvas*, const Allocation&,
                                                Extension&);
        virtual void draw(Canvas*, const Allocation&) const;
private:
        Color* black;
        Color* grey;
        Brush* dick;
        Brush* duenn;
};

#endif
```

Diese Klasse wird dafür verantwortlich sein, die gewünschte Linie von links unten nach rechts oben zu zeichnen. Dazu werden die drei **Glyph**–Memberfunktionen „**request**“, „**allocate**“ und „**draw**“ neu definiert.
Das File **linie.c** leistet die eigentliche Lösung der Aufgabenstellung. Es enthält die Implementierung der Klasse **Linie**:

```
#include "header.h"

Linie::Linie() : Glyph() {
        black = new Color(0.0f,0.0f,0.0f);
        grey  = new Color(0.5f,0.5f,0.5f);
        duenn = new Brush(1.0);
        dick  = new Brush(5.0);
        Resource::ref(black);
        Resource::ref(grey);
        Resource::ref(duenn);
        Resource::ref(dick);
}
```

```
Linie::~Linie() {
        Resource::unref(black);
        Resource::unref(grey);
        Resource::unref(duenn);
        Resource::unref(dick);
}

void Linie::request(Requisition &req) const {
        Requirement rx(250.0,150.0,100.0,0.0f);
        Requirement ry(150.0,50.0,0.0,0.0f);

        req.require(Dimension_X,rx);
        req.require(Dimension_Y,ry);
}

void Linie::allocate(Canvas* c,const Allocation &a,
                     Extension &ext) {
        ext.set(c, a);
}

void Linie::draw(Canvas *c,const Allocation &a) const {
        Coord x1 = a.left();
        Coord y1 = a.bottom();
        Coord x2 = a.right();
        Coord y2 = a.top();

        c->new_path();
        c->move_to(x1,y1);
        c->line_to(x2,y2);
        c->close_path();
        c->stroke(black,duenn);
}
```

Der Konstruktor ruft den Konstruktor der Basisklasse **Glyph** auf, und legt dann die zu verwendenden Farben und Pinsel fest. Hier könne Sie schön sehen, wie ein *shared object* referenziert werden muß. Die im Manual von Stefan Mayer [IV3.0] verwendete Definition von Farbe und Pinsel in der **draw** – Funktion selbst ist eine Speicherplatzverschwendung, da bei jedem Aufruf von `Linie::draw` neue Objekte erzeugt werden, was ja recht häufig passiert.

Die Dereferenzierung der *shared objects* übernimmt der Destruktor der Klasse **Linie**.

Mit den vorherigen Informationen ist jetzt die Memberfunktion `Linie::request` leicht zu verstehen. Zunächst wird je ein **Requirement**–Objekt für jede Dimension erzeugt. Die *natürliche Größe* wird in x–Richtung auf **250.0** (in **Coords**) festgelegt. Das Objekt soll in x–Richtung um **150** Koordinateneinheiten *dehnbar* und um **100** Koordinateneinheiten *stauchbar* sein, d.h. es ist maximal **400** und minimal **150** Koordinateneinheiten breit. Entsprechendes gilt für die y–Richtung. Den Koordinatenursprung legen wir auf links unten durch Setzen des **align** – Parameters bei beiden Dimensionen auf **0.0f**. Mit **r.require**(...) werden die beiden **Requirement**–Objekte rx und ry in das **Requisition**–Objekt **r** eingebaut.

In `Linie::allocate` wird der **Extension** die berechnete **Allocation** durch ext.set(...) zugewiesen.

Nun fehlt nur noch die Beschreibung von `Linie::draw`. Diese Routine bewirkt die eigentliche Ausgabe auf dem Bildschirm. Die beiden Memberfunktionen `Linie::request` und `Linie::allocate` waren dagegen für die Erledigung von *Verwaltungsaufgaben* zuständig.

Zur Erinnerung: `Linie::draw` wird vom System im Anschluß an `Linie::allocate` mit denselben **Canvas***– und **const Allocation&**–Parametern aufgerufen. Den Rumpf von `Linie::draw` sollten Sie jetzt schon ohne meine Hilfe verstehen können. Trotzdem möchte ich nochmals kurz den Sourcecode erklären.

Anfangs werden die Koordinatenpaare x1 und y1 bzw. x2 und y2 mit den Koordinaten der linken unteren bzw. rechten oberen Ecke des vom System übergebenen **Allocation**–Objekts besetzt. Schließlich wird noch die schwarze Linie von der linken unteren in die rechte obere Ecke mittels Pfaddeklaration und nachfolgendem **Canvas::stroke** gezeichnet.

Das eigentliche Hauptprogramm ist in dem File **main.c** enthalten:

```
#include "header.h"

int main(int argc, char **argv) {

  Session *session = new Session("glyphtest",argc,argv);
  WidgetKit& kit = *WidgetKit::instance();
  Linie *linie = new Linie();
  Background *backg = new Background(linie, kit.background());
  Window *window = new ApplicationWindow(backg);

  return session->run_window(window);
}
```

Die Struktur von **main.c** ist derjenigen des Beispiels aus Kapitel 1 sehr ähnlich. Es wird wie in dem Beispielprogramm aus Kapitel 1 (**hello.c**) ein **ApplicationWindow** erzeugt, mittels **session–>run_window** auf den Bildschirm gebracht und eine Eingabeabfrageschleife gestartet. Im Unterschied zu **hello.c** wird dem Fenster hier aber kein Zeiger auf ein **Label**–Objekt, sondern ein Zeiger auf ein Objekt unserer eigenen Klasse **Linie** übergeben. Um auf einen geeigneten Hintergrund (*Standard – weiß*) zeichnen zu können, packen wir unseren **Linien – Glyph** vorher noch in ein **Background**–Objekt, welches wir dann dem **ApplikationWindow** übergeben. Zur Erzeugung eines Standard – Hintergrunds verwenden wir wieder eine **instance** des **WidgetKits** mit der Memberfunktion `WidgetKit::background`.

Um das angegebene Programm zu kompilieren, kopieren Sie sich das **Imakefile** aus Abschnitt 1.5 in das Verzeichnis, in dem **header.h**, **linie.h**, **linie.c** und **main.c** stehen. Ändern Sie das **Imakefile** auf folgenden Inhalt:

```
SPECIAL_IMAKEFLAGS = -DUseInstalled -DTurnOptimizingOn=0

AOUT = linie

Use_libInterViews()

ComplexProgramTarget(a.out)

MakeObjectFromSrc(linie)
MakeObjectFromSrc(main)
```

Erzeugen Sie aus diesem **Imakefile** durch „**ivmkmf**“ ein **Makefile**. Versuchen Sie nun, das Programm zu übersetzen und auszuführen.

2.4 Anregungen

Zum Abschluß dieses Kapitels will ich Ihnen noch einige Anregungen zum Experimentieren geben.

Als gute Übung könnte der Konstruktor der Klasse **Linie** parametrisiert werden. Als Argumente können z.B. die acht Parameter angegeben werden, die in `Linie::request` zum Aufbau des **Requisition**–Objekts verwendet werden. Diese Argumente sollten im Konstruktor `Linie::Linie` in neu einzuführende private Membervariablen abgespeichert werden. Die Memberfunktion `Linie::request` kann dann diese Variablen verwenden. Nun können Sie in **main.c** bequem die

Größe des erzeugten **Linie**–Objekts steuern. Probieren Sie unterschiedliche Parametersätze aus und beobachten Sie, wie sich das Verhalten des Bildschirmfesters ändert, wenn Sie eine andere *natürliche Größe* oder *Dehnbarkeit* bzw. *Stauchbarkeit* angeben.

Genauso können Sie andere „**draw**“ – Funktionen verwenden. Versuchen Sie sich an `Canvas::line` und `Canvas::rect` oder `Canvas::rect_fill`; setzen Sie damit kleine oder größere Punkte auf den **Canvas** und versuchen Sie ruhig, mathematische Funktionen darzustellen. Sie werden dann schnell lernen, Polygonzüge durch Pfadlisten zu zeichnen.

Kapitel 3

Eingabe – Erkennung und Verarbeitung

Nachdem Sie im letzten Kapitel die Grundlagen der Bildschirmausgabe kennengelernt haben, soll nun die Behandlung von Benutzereingaben besprochen werden. Moderne Graphikoberflächen sind interaktiv, d.h. Maus- oder Tastatureingaben werden bearbeitet und die resultierenden Änderungen sofort am Bildschirm angezeigt.

Die Struktur dieses Kapitels gleicht der des Kapitels 2. Nach der Definition einer kleinen Aufgabenstellung und der Erläuterung der benötigten *InterViews* – Klassen schließt sich die Implementierung in Beispiel 3 an, und zwar auf Seite 47.

3.1 Aufgabenstellung

Um das Beispielprogramm so knapp und leicht verständlich wie möglich zu halten, wird hier keine völlig neue Aufgabe gestellt. Vielmehr wird das Beispiel des vorhergehenden Kapitels um eine einfache Eingabemöglichkeit erweitert. Bisher wird in dem Bildschirmfenster nur eine diagonale schwarze Linie dargestellt. Das Programm soll nun durch Drücken einer der drei Tasten „**d**“, „**s**“ oder „**w**“ dazu veranlaßt werden, eine diagonale, senkrechte oder waagrechte Linie zu zeichnen. Außerdem wollen wir einen Weg beschreiben, das Programm auf geordnetem Wege wieder zu verlassen.

3.2 Notwendige InterViews – Klassen

In diesem Abschnitt werden Sie die drei Klassen **Event**, **InputHandler** und **Hit** kennenlernen. Außerdem wird die Klasse **Glyph** unter einem anderem Aspekt als bisher behandelt.

3.2.1 Event

Objekte dieser Klasse repräsentieren Eingabeereignisse wie z.B. Maus- oder Tastaturklicks. Der *InterViews* – Programmierer braucht sich um Verwaltungsaufgaben wie Erzeugen oder Löschen von **Event**–Objekten nicht zu kümmern. Dies wird vom System erledigt. Für jedes **Display**, das eine *InterViews* – Applikation geöffnet hat (in der Regel nur das Defaultdisplay), verwaltet das System einen Strom von **Events**. Tritt nun z.B. ein Mausklick auf, so wird das entsprechende **Event** automatisch erzeugt und in den Strom eingefügt. Gleichzeitig versucht das System, die **Events** dieses Stroms der *InterViews* – Anwendung zur Verarbeitung zu übergeben. Die Übergabe und die Verarbeitung der **Events** werden bei der Diskussion der Klassen **Glyph** und **InputHandler** besprochen. Zunächst soll aber die Klasse **Event** anhand ihrer Deklaration genauer untersucht werden:

```
typedef unsigned int EventType;
typedef unsigned int EventButton;

class Event {
  public:
    enum {undefined, motion, down, up, key, other_event};
    enum {none, any, left, middle, right, other_button};
    virtual EventType type() const;
    virtual EventButton pointer_button() const;
    virtual Coord pointer_x() const;
    virtual Coord pointer_y() const;
    virtual unsigned char keycode() const;
    virtual unsigned int mapkey(char*, unsigned int len) const;
    ...
};
```

Jedem erzeugten **Event** wird zur Beschreibung ein **EventType** und ein **EventButton** mitgegeben. Die **EventTypes** „**motion**“, „**down**“ und „**up**“ entstehen beim Bewegen der Maus oder drücken bzw. Loslassen einer der Mausknöpfe. Ein **key**–**Event** tritt auf, wenn auf der Tastatur eine Taste betätigt wird. Alle anderen **Events** werden als **other_event** bezeichnet (Dazu gehören z.B. **Events**, die zur Fensterverwaltung unter X nötig sind.), und sind für uns nicht interessant. Elemente des Typs **EventButton** spezifizieren, welche Maustaste gedrückt wurde. (bei einem anderen als einem MausButton–**Event** steht der **EventButton** auf „**none**“.) `Event::left` zeigt beispielsweise an, daß die linke Maustaste betätigt wurde. Die Bedeutung der restlichen Elemente dieses Aufzählungstyps ist offensichtlich.

Die Memberfunktion `Event::type` liefert den **EventType** des **Events** zurück, `Event::pointer_button` den **EventButton**. `Event::pointer_x` und `Event::pointer_y` geben die Position des Mauszeigers zu dem Zeitpunkt an, an dem das **Event** auftrat. Die Angabe der Position erfolgt relativ zur linken unteren Ecke des Fensters, zu dem das **Event** gehört. Ist das **Event** vom Typ „**key**“, so liefert `Event::keycode` den maschinenspezifischen Tastaturcode der gedrückten Taste. Da dieser Code Maschinenabhängig ist, wird noch die Routine `Event::mapkey` bereitgestellt. Sie übersetzt den intern gespeicherten Tastaturcode in einen String. Dazu werden ihr als erstes Argument die Adresse eines Buffers und als zweites Argument dessen Länge übergeben. Als Resultat liefert `Event::mapkey` die Länge des Strings zurück, den sie im Buffer abgelegt hat. Normalerweise ist diese Länge „1“ und in **buffer[0]** findet sich dann das Zeichen, das zu der gedrückten Taste gehört.

3.2.2 Glyph

Als nächstes wollen wir uns noch einmal unserer Basisklasse **Glyph** zuwenden. Sie ist nicht nur für die Ausgabe auf den Bildschirm zuständig, sondern dient auch der Verarbeitung von Eingabeereignissen. Wenn während einer *InterViews* – Anwendung ein Eingabeereignis, ein sogenannter **Event**, auftritt, wird zunächst das zuständige Bildschirmfenster bestimmt (In *InterViews* also ein **Window** bzw. eine davon abgeleitete Klasse). Sie wissen ja bereits, daß ein **Window**–Objekt im Konstruktor einen Zeiger auf ein **Glyph**–Objekt erhält, das das Aussehen und Verhalten des Fensters bestimmt. Da das **Window** das vom System übergebene **Event** nicht interpretieren kann, leitet es das Ereignis an seinen **Glyph** weiter. Bis zu diesem Punkt laufen alle eben beschriebenen Aktionen automatisch ab. Der Teil unserer *InterViews* – Applikation, den wir programmieren müssen, wird erst dann aktiv, wenn dem **Glyph** das **Event** übergeben wird. Diese Übergabe erfolgt durch Aufruf der Memberfunktion `Glyph::pick`, deren Schnittstelle folgende Gestalt hat:

```
class Glyph : public Resource {
  public:
    virtual void pick(Canvas*,const Allocation&,
                      int depth, Hit&);
    ...
};
```

Die ersten beiden Parameter von `Glyph::pick` sind dieselben wie bei der Memberfunktion `Glyph::draw`. Der dritte Parameter „**int depth**“ zeigt dem **Glyph** an, auf welcher Stufe der Glyphhierarchie er sich befindet. Wurde beispielsweise

ein Bildschirmfenster durch

```
Window *w = new ApplicationWindow(new Linie());
```

erzeugt, so wird `Linie::pick` vom System mit **depth=0** aufgerufen. Entstand das Fenster dagegen mittels

```
Window *w = new ApplicationWindow(
              new Background(new Linie(),...));
```

so wird `Linie::pick` von `Background::pick` mit **depth=1** aufgerufen. Der vierte Parameter „**Hit&**“ von **pick** wird am Ende dieses Abschnitts auf Seite 44 näher erklärt, er ist für uns hier nicht weiter von Belang.

`Glyph::pick` ist neben **request**, **allocate** und **draw** die wichtigste Memberfunktion von **Glyph**. Wie diese drei Funktionen ist auch **pick** standardmäßig mit leerem Rumpf vordefiniert und müßte vom Programmierer neu implementiert werden, zum Glück haben uns die Entwickler der Version 3.1 von *InterViews* hier jedoch mit der Schaffung der Klasse **InputHandler** diese Arbeit abgenommen; Wie, soll im nachfolgenden behandelt werden.

Es gibt primitive **Glyphs**, **MonoGlyphs** *und zusammengesetzte Glyphs, sogenannte* **PolyGlyphs**. *Primitive* **Glyphs** *(z.B.* **Labels***) enthalten keine weiteren* **Glyphs**, *während* **MonoGlyphs** *genau einen* **Glyph** *enthalten.* **PolyGlyphs** *können mehrere* **Glyphs** *umfassen.* Wie wir bisher gesehen haben, z.B. bei einem **Background–Glyph**, können **MonoGlyphs** mithilfe des Konstruktors auch ineinander verschachtelt werden, indem sich immer wieder ein anderer **Glyph** um den vorherigen legt, und dabei Funktionen für diesen übernimmt.

In der Regel wird an das **Window**–Objekt demnach fast immer ein Zeiger auf die Wurzel einer ganzen Glyphhierarchie übergeben, die aus verschiedensten **Glyphs** aufgebaut ist. Das System ruft in diesem Fall `Glyph::pick` auf dieser Wurzel auf. Die **pick**–Routinen der **MonoGlyphs** und der **PolyGlyphs** geben diesen Aufruf an die **pick**–Routinen ihrer Komponenten weiter, bis die Blätter der Glyphhierarchie, die primitiven **Glyphs**, erreicht sind.

3.2.3 InputHandler

In älteren *InterViews* – Versionen war es noch nötig, die gesamte Erkennung und Behandlung von Eingabeereignissen explizit auszuprogrammieren. Seit Version 3.1 steht die Klasse **InputHandler** zur Verfügung, mit der Reaktionen auf bestimmte Ereignisse (**Events**) sehr einfach implementiert werden können. Bei **InputHandler** handelt es sich wieder um einen **MonoGlyph**, der sich um den eigentlichen **Glyph** legt; nur daß er die Eingabeereignisse abfängt und verarbeitet. Die HeaderDatei finden wir unter <**InterViews/input.h**>, im folgenden die für uns im Moment wichtigen Teile der Klasse:

```
class InputHandler : public MonoGlyph {
  public:
    InputHandler(Glyph*, Style*);
    virtual Style* style() const;
    virtual void pick(Canvas*, const Allocation&,
                                int depth, Hit&);
    virtual void move(const Event&);
    virtual void press(const Event&);
    virtual void drag(const Event&);
    virtual void release(const Event&);
    virtual void keystroke(const Event&);
    virtual void double_click(const Event&);
    virtual void redraw() const;
    ...
};
```

Der Konstruktor verlangt einen **Glyph**, seinen sogenannten *Body*, also den **Glyph**, auf dem die eigentlichen Aktionen stattfinden, und ein **Style**–Objekt. Auf das **Style**–Objekt kann mittels der Memberfunktion `InputHandler::style` zugegriffen werden. Die Memberfunktion `InputHandler::pick` kennen wir schon vom Standard – **Glyph**, hier wird dem **Glyph** der **Event** vom System übergeben. Die nachfolgenden Memberfunktionen, wie auch „**pick**“, sind virtuelle Memberfunktionen, d.h. die von **InputHandler** abgeleitete Klasse unserer Applikation muß hier ausprogrammiert werden, falls zu den angegebenen **Events** eine Reaktion erfolgen soll. Dabei ruft der **InputHandler** selbständig jeweils die zugeordnete Memberfunktion auf, also:

- Bewegen der Maus – `InputHandler::move`
- Drücken eines Mausknopfes – `InputHandler::press`
- Ziehen der Maus, während ein Mausknopf gedruckt ist – `InputHandler::drag`
- Loslassen eines Mausknopfes – `InputHandler::release`
- Doppelklick auf einen Mausknopf – `InputHandler::double_click`
- Drücken einerTaste auf dem Keyboard – `InputHandler::keystroke`

Bei automatischen Aufruf der Funktion wird gleichzeitig der aktuelle **Event** zur Nachbearbeitung mit übergeben. So können Sie beispielsweise in der Mem-

berfunktion `InputHandler::press` explizit den **EventButton** der gedrückten Maustaste aus dem **Event** auslesen, oder in `InputHandler::keystroke` den gedrückten Buchstaben herausfinden.

Mit diesen Routinen sind eigentlich die wichtigsten **Events** für die Eingabe Aussortiert. Wer noch mehr benötigt, kann dies in der Funktion `InputHandler::pick` auf alle anderen **Events** implementieren. Was die Klasse **InputHandler** bietet, ist praktisch die automatische Übernahme aller Aussortier- und Zuordnungsvorgänge, so daß Sie sich nur mehr in einer einfachen Funktion mit dem **Event** speziell befassen müssen, und die Reaktion darauf an einer einzigen Stelle implementieren können.

3.2.4 Hit

Bisher wurde immer davon gesprochen, daß einem **Glyph** durch Aufruf der Memberfunktion „**pick**“ ein **Event** zur weiteren Verarbeitung übergeben wird. `Glyph::pick` hat aber keinen Parameter vom Typ **Event**. Das **Event** wird vielmehr in dem Parameter „**Hit&**“ an „**pick** übertragen. `Glyph::pick` entnimmt dem **Hit**–Objekt die Eventinformation, wertet sie aus und trägt in das **Hit**–Objekt Informationen ein, die dem System mitteilen, ob bzw. wie der **Glyph** auf das **Event** reagieren will. Das **Hit**–Objekt wird demzufolge als transienter Parameter verwendet. Im weiteren wird die Klasse **Hit** genauer beschrieben:

```
class Hit {
  public:
    Hit(const Event*);
    Hit(Coord x, Coord y);
    Hit(Coord left, Coord bottom, Coord right, Coord top);
    virtual const Event* event() const;
    virtual void target(int depth, Glyph*,
                        GlyphIndex, Handler* = nil);
    ...
};
```

In unseren Beispielen werden wir **Hit**–Objekte nicht selbst erzeugen. Trotzdem soll kurz auf die beiden abgebildeten Konstruktoren eingegangen werden. Der erste ist klar, er erhält als Parameter einen Zeiger auf das **Event**, mit dem das **Hit**–Objekt konstruiert werden soll. Darüberhinaus ist es aber auch möglich, ein **Hit**–Objekt mit einem Rechteck zu erzeugen. Die „**pick**“–Routine wird nämlich nicht nur vom System zur Übermittlung eines **Events** aufgerufen, sondern auch vom Programmierer selbst, um feststellen zu lassen, ob ein **Glyph** eine bestimmte Rechtecksfläche schneidet. Diese beiden Aufgaben sind sehr ähnlich und können

mit den entsprechenden Fallunterscheidungen leicht gemeinsam bei der Implementierung der „**pick**"–Routine realisiert werden. Wir werden allerdings nur die Verarbeitung von **Events** programmieren.

Die Memberfunktion `Hit::event` liefert einen Zeiger auf das **Event**, mit dem das **Hit**–Objekt erzeugt wurde oder „**nil**", falls der Rechteck–Konstruktor verwendet wurde.

Mittels der Routine `Hit::target` kann ein **Glyph** Eintragungen in einem **Hit**–Objekt vornehmen. Wurde das **Hit**–Objekt mit einem **Event** konstruiert, so wird dem System auf diesem Weg mitgeteilt, daß der **Glyph** auf das **Event** reagieren will. Der erste Parameter von „**target**" gibt die Hierarchiestufe des eintragenden **Glyphs** an. Er wird im Regelfall direkt aus dem Argument „**depth**" von `Glyph::pick` übernommen. Als zweites Argument wird ein Zeiger auf den **Glyph** selbst („**this**") angegeben. Damit erklärt der **Glyph**, daß er an dem **Event** interessiert ist. Als drittes Argument wird ein Index angegeben, bei primitiven **Glyphs** „**0**". Als viertes Argument wird ein Zeiger auf ein **Handler**–Objekt übergeben. Dieses **Handler**–Objekt legt fest, welche Aktionen als Reaktion auf das **Event** ausgeführt werden sollen. Natürlich könnten diese Aktionen gleich in `Glyph::pick` angegeben werden. Dennoch sollten zu diesem Zweck **Handler** verwendet werden, weil dieser Mechanismus in *InterViews* Vorteile bietet. So ist es beispielsweise möglich, daß mehrere **Glyphs** einer Glyphhierarchie auf ein **Event** reagieren wollen und ein entsprechendes **Handler**–Objekt in das **Hit**–Objekt eintragen. Das System kann nun daraus entsprechend bestimmten Regeln einen geeigneten **Handler** auswählen und ausführen.

Wird dagegen ein **Hit**–Objekt mittels des Rechteck–Konstruktors erzeugt, so teilt der **Glyph** durch `Hit::target` mit, daß er beim Zeichnen die angegebene Rechtecksfläche überschneiden würde. So erhält die Routine, die `Glyph::pick` auf der Wurzel einer Glyphhierarchie aufruft, in dem **Hit**–Objekt eine Hierarchie von **Glyphs**, die das Rechteck zumindest teilweise überdecken. In diesem Fall braucht in `Hit::target` natürlich kein **Handler** angegeben werden. Die übrigen Parameter von „**target**" haben eine ähnliche Bedeutung wie bei einem **Hit**–Objekt, das mit einem **Event** erzeugt wurde.

Die Klasse **Hit** war vor allem in Früheren Versionen von *InterViews* von Bedeutung, als es die Einfache Behandlung der **Events** mittels **InputHandler** noch nicht gab. Falls Sie nur einfache Programme schreiben, werden Sie diese Klasse vielleicht nie benötigen. Falls Sie aber doch auf die oben genannten speziellen Funktionen von **Hit** zurückgreifen wollen, dann haben Sie jetzt zumindest das Grundverständnis erlangt.

Sie sollten aber auf jeden Fall auch noch die Deklaration und Syntax der Klasse **Handler** *im Referenzmanual [MAN3.1] nachschlagen, Diese ist verschieden von der Klasse InputHandler !*

3.2.5 Zusammenfassung

Zum Abschluß dieses Abschnitts möchte ich noch einmal die Schritte zusammenfassen, die beim Auftreten eines Eingabeereignisses in einer *InterViews* – Applikation ablaufen:

1. Das System bestimmt das Fenster, zu dem das Ereignis gehört.
2. Das System ruft auf dem Wurzelglyph des Fensters `Glyph::pick` mit einem **Hit**–Objekt auf, das mit dem entsprechenden **Event** erzeugt wurde.
3. Die Glyphhierarchie wird mit sukzessiven **pick**–Aufrufen durchlaufen. Dabei werden evtl. mehrere **InputHandler**–Objekte in das **Hit**–Objekt eingetragen.
4. Das System wählt einen geeigneten **InputHandler** aus dem **Hit**–Objekt aus, und führt darauf `InputHandler::pick` aus.
5. Der **InputHandler** erkennt die Art des **Events**, übergibt ihn an die zuständige Memberfunktion und führt diese aus (Falls implementiert).

3.3 Beispiel 3

Auch bei diesem Beispiel werden Sie feststellen, daß die eigentliche Implementierung wesentlich einfacher als erwartet ist. Die theoretischen Teile fallen in diesem Tutorial deshalb so umfangreich aus, weil Sie nicht nur die Beispiele verstehen sollen, sondern nach dem Durcharbeiten auch in der Lage sein sollen, eigene *InterViews* – Programme zu schreiben. Der Code ist in die sechs Files **header.h**, **linie.h**, **linie.c**, **selector.c**, **selector.h** und **main.c** aufgeteilt. Die Datei **header.h** wurde um wenige Includedateien erweitert:

```
#ifndef header_h
#define header_h


#include <IV-look/kit.h>
#include <InterViews/session.h>
#include <InterViews/window.h>
#include <InterViews/background.h>
#include <InterViews/color.h>
#include <InterViews/brush.h>
#include <InterViews/event.h>
#include <stream.h>

#include "linie.h"
#include "selector.h"

#endif
```

Das Hauptprogramm **main.c** unterscheidet sich nur minimal von dem aus Beispiel 2. Neu Hinzugekommen ist nur der Aufruf des Standard – **Styles** und der **InputHandler Selector**, dem unserer **Linie – Glyph** als *Body* im Konstruktor übergeben wird. Anschließend wird um den **InputHandler** der **Background** gelegt; In diesem Falle wäre natürlich die Umgekehrte Verschachtelung der **MonoGlyph**'s auch möglich. Dementsprechend hat **main.c** nun folgende Gestalt:

```
#include "header.h"

int main(int argc, char **argv) {

   Session *session = new Session("Example",argc,argv);
   Style* style = session->style();
   WidgetKit& kit = *WidgetKit::instance();
   Selector *handy = new Selector(new Linie(), style);
   Background *backg = new Background(handy, kit.background());
   Window *window = new ApplicationWindow(backg);

   return session->run_window(window);
}
```

linie.h enthält folgende Deklarationen:

```
#ifndef linie_h
#define linie_h

class Linie : public Glyph {
public:
        Linie();
        virtual ~Linie();
        virtual void request(Requisition&) const;
        virtual void allocate(Canvas*,const Allocation&,
                                              Extension&);
        virtual void draw(Canvas*,const Allocation&) const;

        void set_linetype(char);
private:
        Color *black;
        Brush *duenn;
        char linetype;
};

#endif
```

In **Linie** wird nun eine neue private Membervariable „**linetype**“ verwendet. Sie gibt an, ob eine diagonale, waagrechte oder senkrechte Linie gezeichnet wird.

Die Implementation der Klasse **Linie** ist wiederum in der Datei **linie.c** enthalten:

```
#include "header.h"

Linie::Linie() : Glyph() {
        black = new Color(0.0f, 0.0f, 0.0f);
        duenn = new Brush(1.0);
        Resource::ref(black);
        Resource::ref(duenn);
        linetype = 'd';
}

Linie::~Linie() {
        Resource::unref(black);
        Resource::unref(duenn);
}

void Linie::set_linetype(char type) {
        linetype = type;
}

void Linie::request(Requisition &req) const {
        Requirement rx(500.0,300.0,200.0,0.0);
        Requirement ry(500.0,300.0,200.0,0.0);
        req.require(Dimension_X,rx);
        req.require(Dimension_Y,ry);
}

void Linie::allocate(Canvas *c,const Allocation &a,
                                    Extension &ext) {
        ext.set(c, a);
}

void Linie::draw(Canvas *c,const Allocation &a) const {

        Coord x1 = a.left();
        Coord y1 = a.bottom();
        Coord x2 = a.right();
        Coord y2 = a.top();

        c->new_path();
        switch(linetype) {
```

```
        case 'd':
         c->move_to(x1,y1);
         c->line_to(x2,y2);
         break;
        case 'w':
         c->move_to(x1,(y1+y2)/2);
         c->line_to(x2,(y1+y2)/2);
         break;
        case 's':
         c->move_to((x1+x2)/2,y1);
         c->line_to((x1+x2)/2,y2);
         break;
        }
        c->close_path();
        c->stroke(black, duenn);
}
```

Wir initialisieren „**linetype**“ im Konstruktor mit „**d**“, und damit die Linie mit der diagonalen Darstellung, um einen undefinierten Zustand zu verhindern. In `Linie::request` sind leicht veränderte Werte für die Requisition angegeben. Wie schon in Beispiel 2 werden in `Linie::draw` (x1,y1) bzw. (x2,y2) mit den Koordinaten der linken unteren bzw. rechten oberen Ecke besetzt. Der Abschnitt zur Erzeugung der Linie wird gemäß der Aufgabenstellung um eine Fallunterscheidung erweitert: Abhängig vom Inhalt der Membervariablen „**linetype**“ wird eine diagonale, waagrechte oder senkrechte Linie gezeichnet. Die diagonale Linie erstreckt sich wie bisher von links unten nach rechts oben, die waagrechte von links mitte bis rechts mitte und die senkrechte von mitte unten bis mitte oben.

Im folgenden die Headerdatei **selector.h**, die die Deklaration unserer von **InputHandler** abgekeiteten Klasse enthält:

```
#ifndef selector_h
#define selector_h

class Selector : public InputHandler {
public:
        Selector(Glyph*, Style*);

        virtual void keystroke(const Event&);
        virtual void press(const Event&);
};

#endif
```

Wir leiten Selector von InputHandler ab und implemetieren die Funktionen `Selector::keystroke` und `Selector::press`, da auf Mausklicks und auf Tastatureingaben reagiert werden soll.
Die Implementierung der Funktionen steht in **selector.c** :

```
#include "header.h"

Selector::Selector(Glyph* g, Style* s) : InputHandler(g, s) {
        ((Linie*)body())->set_linetype('d');
}

void Selector::keystroke(const Event& e) {
    const int buflen = 10;
    char buffer[buflen];

    int l = e.mapkey(buffer,buflen);
    if ( buffer[0]=='d' || buffer[0]=='w' || buffer[0]=='s' ) {
       ((Linie*)body())->set_linetype(buffer[0]);
    }
    else
     if ( buffer[0]=='q' )
        Session::instance()->quit();
    redraw();
    InputHandler::keystroke(e);
}
```

```
void Selector::press(const Event& e) {
   const int LEFTMOUSE = 2;
   const int MIDDLEMOUSE = 3;
   const int RIGHTMOUSE = 4;

   if (e.pointer_button()==LEFTMOUSE) {
      cerr << "linker Mausbutton gedrueckt, ";
      cerr << "auf den Koordinaten : ";
      cerr << "x = " << e.pointer_x() << " und ";
      cerr << "y = " << e.pointer_y() << "\n";
   }
   InputHandler::press(e);
}
```

Der Konstruktor von **Selector** übergibt den *Body*–**Glyph** und **Style** auch der Basisklasse, ansonsten wählen wir als Startdarstellung die diagonale Linie, indem wir auf unseren **Linie – Glyph set_linetype('d')** ausführen. Da **Selector** von **InputHandler** und damit auch von **MonoGlyph** abgeleitet ist, können wir auf den übergebenen Glyph mittels der Memberfunktion `MonoGlyph::body` zugreifen. Um jedoch auch **Linie** – spezifische Memberfunktionen auf diesem, standardmäßig nur als Glyph definierten *Body* auszuführen, müssen wir eine Typenkonvertierung nach **Linie*** vornehmen, da sonst der Compiler – zu Recht – diesen Unterschied verweigert.

Die Ausprogrammierung von `Selector::keystroke` zieht ihre Schwierigkeit eigentlich nur aus dem Eingabepuffer, der dem **Event** mitgegeben wird.

Hier wird durch `Event::mapkey` mit Hilfe der Lokalen Variablen „buffer“ die gedrückte Taste ermittelt. Normalerweise ist hier sowieso nur ein einziger Buchstabe von Interesse, da wir ja auf einen einzelnen Tastendruck reagieren wollen. Der Tastaturpuffer könnte jedoch auch mehrere Zeichen enthalten, deswegen hier gleich eine „buffer“–Größe von 10. Falls „**d**“, „**w**“ oder „**s**“ gedrückt wurde, wird „**linetype**“ entsprechend gesetzt. Nun soll aber auch die Linie gemäß dieser Einstellung neu gezeichnet werden. Dazu darf aber nicht einfach `Linie::draw` aufgerufen werden. Der **InputHandler** nimmt uns hier alle Arbeit ab; wir rufen `InputHandler::redraw` auf, bewirken damit ein Neuzeichnen des enthaltenen **Glyphs** und die Darstellung der Änderung auf dem Bildschirm.

Schließlich wird der Taste „**q**“ ebenfalls eine besondere Bedeutung zugewiesen. Durch die **Session** – Memberfunktion „**quit**“ wird die in **main.c** mittels **session–>run_window** gestartete Abfrageschleife von Eingabeereignissen abgebrochen. Die „**main**“ – Routine kann danach die „**return**“ – Anweisung ausführen und somit ein reguläres Programmende ermöglichen. Damit ist der bis-

her nötige explizite Programmabbruch mittels <CTRL>–C nicht mehr erforderlich. Der für die Ausführung von `Session::quit` benötigte Zeiger auf das globale **Session**–Objekt wird von der statischen Memberfunktion `Session::instance` zurückgeliefert:

```
class Session {
  public:
    static Session* instance();
    ...
};
```

Die Memberfunktion `InputHandler::press` zeigt ein Beispiel, wie auf einen speziellen Tastendruck reagiert werden kann, hier das Drücken des linken Mausknopfes. Das Programm reagiert mit der Ausgabe der Koordinaten, wo sich die Maus zum Zeitpunkt des Knopfdrückens auf dem Display befindet. Die Konstruktion mit den Konstanten **MIDDLEMOUSE**, **LEFTMOUSE** und **RIGHTMOUSE** ist notwendig, da `Event::pointer_button` nur den Index des gerade aktuellen **Buttons** im Aufzählungstypen **EventButton** wiedergibt (der als integer–Zahl dargestellt wird), und die Abfrage ja auch in einer verständlichen und vor allem in einer im Programm lesbaren Form dargestellt werden soll.

Zum Abschluß jeder Funktion, die in unserer Klasse **Selector** die Eingabeereignisse abfängt, sollte noch der Aufruf der Basisklasse **InputHandler** mit derselben Memberfunktion und demselben **Event** erfolgen, damit die Bearbeitung korrekt weitergeleitet werden kann, und keine **Events** verloren gehen. Hier müssen wir also am Ende von `Selector::keystroke` auch noch `InputHandler::keystroke(e)` aufrufen und genauso am Ende der Implementierung von `Selector::press` noch `InputHandler::press(e)`. Die Ursache liegt in der Art der Abarbeitung des Eingabestroms in *InterViews*, dieses Detail soll aber hier nicht weiter besprochen werden.

3.4 Anmerkungen

Die Erstellung des zugehörigen **Imakefiles** sollte nun keine Probleme mehr bereiten. Übersetzen Sie das Programm und versuchen Sie eventuell, weiteren Tasten des Keyboards eine bestimmte Bedeutung zuzuordnen (z.B. Farbe oder Pinselgröße). Sie könnten auch andere Mausereignisse abfragen und damit beispielsweise einen kleinen Editor zum Zeichnen von Linien programmieren.

Kapitel 4

Menüs

Mittlerweise haben Sie gesehen, wie in *InterViews* Bildschirmausgaben erzeugt werden, und wie die Eingabeverarbeitung funktioniert. Nun ist es aber sehr umständlich, ein Programm nur über Tastendruck zu steuern. Darüber hinaus ist es oftmals schwierig für einen neuen Benutzer, sich die für die Bedienung eines komplexen Programms erforderlichen Tastenkürzel einzuprägen.

Daher haben sich für die Interaktion zwischen Programm und Benutzer auch andere Techniken durchgesetzt. Bekanntestes Beispiel sind die **Pulldownmenüs**, deren Erzeugung unter *InterViews* Gegenstand dieses Kapitels sein soll.

Tastaturkürzel sind deshalb nicht weniger wichtig, so sind in heutigen Menüs oft Standardmäßig (siehe Microsoft Windows) zu den einzelnen Menüpunkten einzelne Tastenkombinationen angegeben, die zum selben Ergebnis führen wie das Anklicken des Menüs. Dies ist bei Benutzern, die mit dem selben Programm sehr lange arbeiten, oft von großen Vorteil, da sie z.B. bei Textverarbeitungen ja sowieso die ganze Zeit auf der Tastatur herumtippen, und die Bedienung über Funktionstasten sowohl eine gewisse Zeitersparnis bietet als auch bequemer ist. Nichtsdestotrotz sind Menüs die einfachste Möglichkeit, einen ungeübten Benutzer an ein neues Programm heranzuführen, und ihn mit der Bedienung vertraut zu machen. Die Kombination beider Techniken mit einem übersichtlichen, logisch und den allgemein üblichen Standards folgendem Aufbau zeichnet heute ein gutes und daher erfolgreiches Programm aus. Das Beispielprogramm 4 finden Sie ab Seite 64.

4.1 Aufgabenstellung

Wie auch in den vorangegangenen Beispielen soll der Programmcode so knapp wie möglich gehalten werden. Deshalb wird hier nur eine ganz einfache Menüstruktur aufgebaut. Das Anklicken eines Menüpunkts wird lediglich die Ausgabe einer Meldung der Form „... **called**“ bewirken.

Die genaue Struktur des Beispielmenüs ist aus Abbildung 4.1 ersichtlich.

Die Menüleiste umfaßt zwei **Pulldownmenüs**. Das erste, das den Titel „**File**“

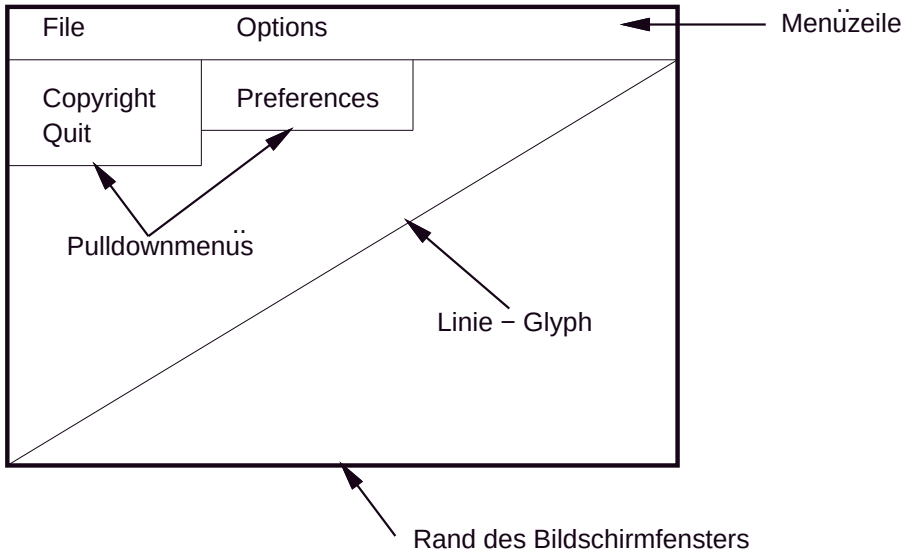


Abbildung 4.1: Menüstruktur des Beispiels

trägt, beinhaltet die Menüpunkte „**Copyright**“ und „**Quit**“. Das **Pulldownmenü** „**Options**“ besteht lediglich aus dem Eintrag „**Preferences**“. Als Hintergrund für die Menüstruktur wurde ein Objekt unserer Klasse **Linie** gewählt. Normalerweise findet sich an dieser Stelle irgendeine graphische Darstellung, die durch das Menü beeinflußt werden kann. Da unser Menü aber nur Demonstrationszwecken dient, kann hier ein beliebiger „Platzhalter“ gewählt werden.

4.2 Notwendige InterViews – Klassen

Bevor wir uns das eigentliche Programm ansehen, werden zunächst die Klassen **ActionCallback**, **WidgetKit**, **Menu** und **LayoutKit** vorgestellt.

4.2.1 ActionCallback

Wenn der Benutzer einen Menüeintrag anwählt, soll eine bestimmte Aktion ausgeführt werden. Eine solche Aktion wird in *InterViews* von einem Objekt der Klasse **Action** repräsentiert:

```
class Action : public Resource {
  protected:
    Action();
    virtual ~Action();
  public:
    virtual void execute() = 0;
};
```

Action ist eine abstrakte Basisklasse. Der Programmierer muß eigene Subklassen von **Action** ableiten. In der Implementation der virtuellen Memberfunktion `Action::execute` wird dann festgelegt, was bei der Ausführung dieser Aktion geschehen soll.

Für viele Anwendungen wird beispielsweise eine Action benötigt, die für ein bestimmtes Objekt einer Klasse eine festgelegte Memberfunktion aufruft. Zu diesem Zweck stellt *InterViews* die von **Action** abgeleitete generische Klasse **ActionCallback**(*Typname*) zur Verfügung:

```
#define declareActionCallback(T) \
typedef void (T::*ActionMemberFunction(T))(); \
class ActionCallback(T) : public Action { \
  public: \
    ActionCallback(T)(T*, ActionMemberFunction(T)); \
\
    virtual void execute(); \
  private: \
    T* obj_; \
    ActionMemberFunction(T) func_; \
};

#define implementActionCallback(T) \
ActionCallback(T)::ActionCallback(T)\
  (T* obj, ActionMemberFunction(T) func) { \
  obj_ = obj; \
  func_ = func; \
} \
\
void ActionCallback(T)::execute() { (obj_->*func_)(); }
```

Da C++ derzeit noch keine generischen Typen erlaubt, erfolgt die Deklaration und Implementierung der Klasse **ActionCallback(***Typname***)** durch vordefinierte Makros. Sei z.B. **TestMenu** ein Klassenname und `TestMenu::copyright` eine Memberfunktion von **TestMenu**.

Dann wird die Klasse **ActionCallback(TestMenu)** durch `declareActionCallback(TestMenu)` deklariert und durch `implementActionCallback(TestMenu)` implementiert. Mittels

```
Action *copyright = new ActionCallback(TestMenu)(this,
                                &TestMenu::copyright);
```

wird eine Action „**copyright**“ erzeugt, die bei `copyright-->execute()` den Aufruf `tm-->copyright()` bewirkt. Das Resultat ist, daß Sie ein **Menü** mit der Erzeugung von speziellen **Actions** ausstatten können, die bei Aufruf im laufenden Programm bestimmte vordefinierte Memberfunktionen starten. Diesen Vorgang nennt man *ActionCallback*, er verbindet sozusagen das **Menü** mit dem übrigen Programmtext. Ähnlich wie **Events** an **Glyphs** weitergereicht werden, werden so **Actions** an einzelne Memberfunktionen weitergegeben; dieser Weg ist zweifellos direkter, weshalb Sie ihn auch erst in ihrem Programm explizit implementieren müssen. Im Beispielprogramm werden wir das **Menü** mit Hilfe dieser **Actions** realisieren.

4.2.2 WidgetKit

Wir wenden uns jetzt nochmals dem *Kit* **WidgetKit** zu, das Sie schon in Kapitel 1 kennengelernt haben. Es stellt ein sehr mächtiges Tool dar, um sehr einfach verschiedene Objekte der Graphischen Oberfläche, interaktiv oder auch nicht, zu erstellen, z.B. **Buttons**, **Labels** oder eben auch **Menüs**. Dabei wurde vor allem Wert auf ein einheitliches Erscheinungsbild (*look and feel*) der Oberflächen gelegt. In der UNIX–Welt sind vor allem **Motif**– und **OpenLook**–Oberflächen weit verbreitet. *InterViews* erfüllt diese Anforderungen mit der Klasse **WidgetKit**.
Die Deklaration der Klasse **WidgetKit** hat folgende Gestalt:

```
class WidgetKit {
  public:
    static WidgetKit* instance();
    virtual Style* style() const;
    virtual const Color* foreground() const;
    virtual const Color* background() const;
    virtual MonoGlyph* outset_frame(Glyph*) const = 0;
    virtual MonoGlyph* inset_frame(Glyph*) const = 0;
    virtual Glyph* label(const char*) const;
    virtual Glyph* chiseled_label(const char*) const;
```

```
    virtual Glyph* raised_label(const char*) const;
    virtual Glyph* fancy_label(const char*) const;

    virtual Menu* menubar() const;
    virtual Menu* pulldown() const;
    virtual MenuItem* menubar_item(const char*) const;
    virtual MenuItem* menu_item(const char*) const;
    virtual MenuItem* menu_item_separator() const;
    virtual Action* quit() const;
    ...
};
```

Was Sie hier sehen, ist wiederum nur ein Bruchteil von **WidgetKit**. Bitte schlagen Sie im Referenzmanual [MAN3.1] nach oder gehen sie durch die Headerdatei **kit.h** im Verzeichnis <**IV–look**>, um die weiteren Funktionen kennenzulernen. Damit können Sie schon noch einiges mehr ausprobieren, als im folgenden gezeigt wird.

Jede *InterViews* – Anwendung darf höchstens ein **WidgetKit**–Objekt erzeugen. Die statische Memberfunktion `WigetKit::instance` liefert einen Zeiger auf ein evtl. bereits vorhandenes **WidgetKit**–Objekt zurück. Existierte zum Zeitpunkt des Aufrufs von „**instance**“ noch kein **WidgetKit**–Objekt, so wird automatisch eines erzeugt und ein Zeiger darauf zurückgegeben.

`WidgetKit::style` stellt das default–**Style** zur Verfügung;
`WidgetKit::foreground` und `WidgetKit::background` haben Sie schon im letzten Beispiel verwendet, sie liefern einfach den Standardwert für die Vorder- und Hintergrundfarbe Ihres Bildschirms zurück.

Die beiden Memberfunktionen `WidgetKit::inset_frame` und `WidgetKit::outset_frame` erzeugen einen **MonoGlyph**, der den übergebenen *Body* mit einem andersfarbigen Rahmen, einem sogenannten „**Bevel**“ versieht, der ihn vor- oder zurückgesetzt erscheinen läßt. Dies ist reine Kosmetik, um einen Glyph 3D–ähnlich von seiner Umgebung abzuheben. In unserem Beispiel werden wir `WidgetKit::inset_frame` verwenden, um den Linie – Glyph als Maloberfläche optisch auszuzeichnen.
*„**Bevels**“ waren in Version 3.0 von InterViews noch eigene Klassen, die aber in Version 3.1 weggelassen wurden; sie sind nun vollständig durch* **WidgetKit**–*Objekte ersetzt. In nachfolgenden Abbildungen beziehe ich mich der Einfachheit halber weiterhin auf „**Bevels**“, gemeint sind damit Objekte, die durch* `WidgetKit::inset_frame`, `WidgetKit::outset_frame` *oder ähnliche Konstruktoren erzeugt wurden.*

`WidgetKit::label` haben wir schon in Beispiel 1 verwendet. Die drei anderen **Label**–Erzeugungs–Funktionen lernen Sie am besten durch Ausprobieren näher kennen, wozu Sie z.B. das Programm von Beispiel 1 modifizieren.

Die Memberfunktion `WidgetKit::quit` liefert einen Zeiger auf eine **Action**, die `Session::instance-->quit()` ausführt. Diese **Action** ist für einen Menüpunkt „**Quit**“ gedacht.

`WidgetKit::menubar` erzeugt eine **Menüleiste**. `WidgetKit::pulldown` erzeugt ein **Pulldownmenü**. Die Funktion `WidgetKit::menu_item` konstruiert ein **Telltale**–Objekt für einen Menüeintrag mit dem übergebenen Text.
Die Klasse **Telltale** *ist von* **MonoGlyph** *abgeleitet und wird für Objekte verwendet, deren Aussehen davon abhängt, ob sie hervorgehoben, angewählt, o.ä. sind.* **Telltales** *eignen sich daher für Menüeinträge oder auch* **Buttons**.
`WidgetKit::menubar_item` erzeugt schließlich einen **Telltale** für einen Eintrag in eine **Menüleiste**.

`WidgetKit::menu_item_seperator` eignet sich zum optischen Trennen zweier **Menu**–Objekte. Dazu später mehr.

4.2.3 Menu

Objekte dieser Klasse dienen als Bausteine für Menüstrukturen. Wir erzeugen solche Objekte nicht selbst, sondern überlassen diese Aufgabe dem **WidgetKit**–Objekt. Dennoch müssen wir uns mit der Klasse **Menu** genauer auseinandersetzen, weil sie die Memberfunktionen bereitstellt, die erforderlich sind, um die Menübausteine zusammenzusetzen:

```
class Menu : public InputHandler {
  public:
    Menu(Glyph*,Style*,float x1,float y1,float x2,float y2);
    virtual void append_item(MenuItem*);
    virtual void prepend_item(MenuItem*);
    ...
};
```

Von den vier „**append_item**“ – Memberfunktionen sind nur die zwei wichtigsten abgebildet. Die erste Form wird dazu verwendet, einen Menüeintrag und die dazugehörige **Action** in ein **Pulldownmenü** einzufügen. Die zweite Form „**prepend_item**“ macht dasselbe, nur wird der Eintrag *vor* den bisherigen eingesetzt.

4.2.4 MenuItem

MenuItems sind die Elementaren Einzelbestandteile eines **Menüs**. Sie stellen die **Buttons** dar, die letztendlich durch ihre Aktivierung das Verhalten des **Menüs** bestimmen, und die die Verbindung zum Programm über **ActionCallbacks** herstellen.

```
class MenuItem {
  public:
    MenuItem(Glyph*, TelltaleState*);
    virtual void action(Action*);
    virtual void menu(Menu*, Window* = nil);
    ...
}
```

Jedes **MenuItem** wird mit einer Aufschrift (**Label**) von **WidgetKit** erzeugt, und muß dann mit seinen Memberfunktionen noch eine **Action** oder ein **Menu** zugewiesen bekommen, die es bei Anwahl selbständig aufruft.

4.2.5 Menü – Erzeugung: Zusammenfassung

Wir können also abschließend zusammenfassen:

- Mittels `WidgetKit::menubar` wird eine neue **Menüleiste** vom Typ **Menu** erzeugt.
- Einzelne **MenuItems** werden mit `WidgetKit::menu_item(Label*)` erzeugt.
- Diesen **MenuItems** wird durch `MenuItem::action` ein **ActionCallback** zugewiesen, den sie bei Aufruf ausführen sollen.
- **Pulldownmenüs**, auch vom Typ **Menu**, werden durch `WidgetKit::pulldown` erzeugt.
- Einzelne **MenuItems** werden in die **Pulldownmenüs** mit `Menu::append_item` eingehängt.
- Für jedes **Pulldownmenü** wird ein eigenes (Ober–) **MenuItem** mit dem **Label**, der den Namen des **Menüs** trägt, erzeugt.
- Das **Pulldownmenü** wird durch `MenuItem::menu` in das (Ober–) **MenuItem** eingetragen. Dies bewirkt statt **ActionCallback** die Darstellung des **Pulldownmenüs**.
- **Pulldownmenüs** werden durch `Menu::append_item` ihres zugehörigen (Ober–) **MenuItems** an die Menüleiste angehängt.

4.2.6 LayoutKit

Wenn eine Menüleiste mit den zugehörigen **Pulldownmenüs** erzeugt wurde, muß sie noch mit dem Menürumpf (in unserem Fall ein **Linie**–Objekt) zusammengesetzt werden. Da eine **Menüleiste** ein Objekt der Klasse **Menu** ist, und diese von **MonoGlyph** abgeleitet ist, stellt auch eine **Menüleiste** wie unser **Linie**–Objekt nur einen speziellen **Glyph** dar. Damit stoßen wir hier erstmals auf die Aufgabe, einen zusammmengesetzten **Glyph** zu erzeugen. Das *Kit* **LayoutKit** wurde dafür geschaffen, uns dazu einige Möglichkeiten anzubieten und einiges an Implementation abzunehmen. Wir wollen deshalb hier auf sie kurz eingehen:

```
class LayoutKit {
  public:
    static LayoutKit* instance();
    virtual PolyGlyph* hbox(
      Glyph* = nil, Glyph* = nil, Glyph* = nil,
      Glyph* = nil, Glyph* = nil, Glyph* = nil,
      Glyph* = nil, Glyph* = nil, Glyph* = nil,
      Glyph* = nil) const;
    virtual PolyGlyph* vbox(
      Glyph* = nil, Glyph* = nil, Glyph* = nil,
      Glyph* = nil, Glyph* = nil, Glyph* = nil,
      Glyph* = nil, Glyph* = nil, Glyph* = nil,
      Glyph* = nil) const;
    virtual Glyph* glue(
      DimensionName, Coord natural, Coord stretch,
      Coord shrink, float alignment) const;
    virtual Glyph* hglue(Coord natural, Coord stretch,
                         Coord shrink) const;
    virtual Glyph* vglue(Coord natural, Coord stretch,
                         Coord shrink) const;
    virtual MonoGlyph* hcenter(Glyph*, float x = 0.5) const;
    virtual MonoGlyph* vcenter(Glyph*, float y = 0.5) const;
    ...
};
```

Wie **WidgetKit** wird auch **LayoutKit** pro **Session** nur einmal erzeugt; der Aufruf der Memberfunktion `LayoutKit::instance` übergibt einen Zeiger auf ein **LayoutKit**, und falls dieses noch nicht existiert, wird es damit automatisch erstellt.

4.2.6.1 Box

Die Memberfunktionen `LayoutKit::hbox` und `LayoutKit::vbox` ordnen die Ihnen übergebenen **Glyphs** in einer „Box“ (engl. Schachtel) nebeneinander bzw. übereinander in der angegebenen Reihenfolge an („**h**“ wie horizontal, „**v**“ wie vertikal). Der Konstruktor erlaubt bis zu zehn Komponenten. Weitere Komponenten können mit `Polyglyph::append` eingefügt werden.

4.2.6.2 Glue

Die einfache unter- bzw. nebeneinander – Anordnung verschiedener **Glyphs** bietet meistens keine befriedigende Ansicht, da ja jeder **Glyph** eine verschiedene Ausdehnung haben kann. Zum Zwecke der besseren Positionierung mehrerer **Glyphs** zusammen wurden Zwischenraum – Füller, sogenannte **Glues** (engl. „Klebstoff“) implementiert, die sich je nach ihren durch den Konstruktor festgelegten Möglichkeiten zwischen den **Glyphs** ausbreiten. Die Deutung der **Coord** – Konstruktorvariablen „**natural**“, „**stretch**“ und „**shrink**“ sollte Ihnen keine Probleme mehr bereiten (siehe Kapitel 2: „**Glyph**“). Diese legen fest, welches Maß die *natürliche Größe* des Zwischenraums hat, und inwieweit er *gedehnt* bzw. *gestaucht* werden kann, falls dies wegen den eingeschlossenen **Glyphs** notwendig wird. **LayoutKit** stellt einen `LayoutKit::hglue` und einen `LayoutKit::vglue` in horizontaler bzw vertikaler Richtung zur Verfügung. Außerdem bietet **LayoutKit** mit `LayoutKit::hcenter` und `LayoutKit::vcenter` in diesen beiden Richtungen sowohl die Möglichkeit, einen **Glyph** innerhalb eines umgebenden Zwischenraums zentriert wiederzugeben, als ihn auch innerhalb diesem mit der zweiten Koordinate zu positionieren (**0** = ganz links, **1** = ganz rechts). Spielen Sie ruhig einmal die Dehnbarkeitsmöglichkeiten der **Glues** mit Beispiel 1 durch, indem Sie den darzustellenden **Label** in **Glues** einpacken, die – mal links – mal rechts – etwas größer sind. Sie werden schnell den Umgang mit diesen variablen Platzfüllern erlernen und deren Nützlichkeit erkennen.

Wenn Sie das Satzsystem TEX kennen, werden Sie feststellen, daß dort für die Layoutberechnung ebenfalls obiges Boxprinzip verwendet wird. Dies ist nicht weiter verwunderlich, da InterViews an der Stanford University entwickelt wird, wo auch Donald Knuth, der Autor von TEX, arbeitet. Die InterViews – Autoren haben das Boxprinzip von Knuths Arbeit weitgehend übernommen (vgl. auch [TeX]).

4.2.7 Rule

Objekte der Klasse **Rule** können in **PolyGlyphs**, z.B. **Menüs**, dazu verwendet werden, Komponenten optisch zu trennen. Ein Objekt der Klasse **Rule**, das in einer „**hbox**“ zwischen zwei **Glyphs** eingefügt wird, erzeugt eine Trennlinie beliebiger Dicke und Farbe zwischen diesen beiden **Glyphs**. Die Deklaration der Klasse **Rule** sieht wie folgt aus:

```
class Rule : public Glyph {
  public:
    Rule(DimensionName, const Color*, Coord thickness);
    ...
};
```

Sie müssen im Konstruktor angeben, ob Sie eine Linie in **Dimension_X** , also horizontal, oder in **Dimension_Y**, also vertikal einfügen möchten; das zweite Argument des Konstruktors gibt die Farbe, das dritte die Breite der Linie an. Analog zu **Rule** gibt es auch noch die Klassen **HRule** und **VRule** mit demselben Konstruktor und Effekt, hier ist nur das erste Argument **DimensionName** im Konstruktor durch den Klassennamen ersetzt; **VRule** zeichnet vertikal, **HRule** horizontal. In **Menüs** bietet sich auch `WidgetKit::menu_item_seperator` an, hier wird eine Standardgröße und Farbe gewählt.

4.3 Beispiel 4

Der Programmcode ist in die sechs Files **header.h**, **linie.h**, **linie.c**, **testmenu.h**, **testmenu.c** und **main.c** aufgeteilt. Die Datei **header.h** ist gegenüber Beispiel 3 um die zusätzlich nötigen *InterViews* – Headerfiles für das **LayoutKit** und die neue Headerdatei **testmenu.h** erweitert worden; **Event** und **Selector** werden in diesem Beispiel nicht benötigt:

```
#ifndef header_h
#define header_h

#include <IV-look/kit.h>
#include <InterViews/session.h>
#include <InterViews/window.h>
#include <InterViews/background.h>
#include <InterViews/color.h>
#include <InterViews/brush.h>

#include <InterViews/layout.h>
#include <stream.h>

#include "linie.h"
#include "testmenu.h"

#endif
```

Die beiden Files **linie.h** und **linie.c** sind gegenüber Beispiel 3 unverändert und können einfach übernommen werden. Neu dagegen ist **testmenu.h**:

```
#ifndef testmenu_h
#define testmenu_h

class TestMenu {
public:
     TestMenu(Glyph*);
     PolyGlyph* compose(WidgetKit&);
     Menu* CreateMenu(WidgetKit&);

     void quit();
     void copyright();
     void preferences();
private:
     Glyph* my_glyph;
};
declareActionCallback(TestMenu)
#endif
```

Auch aus diesem Listing können Sie die Methodik erkennen, mit der in *InterViews* programmiert wird: *Für eigene Objekte, die aus InterViews – Objekten zusammmengesetzt sind oder die die Funktionalität von InterViews – Objekten erweitern, werden eigene Klassen definiert.* **TestMenu** ist die Klasse, in der die Funktionalität für das Erzeugen des gesamten **Menüs** bereitgestellt wird.

Im Konstruktor wird der **Glyph** übergeben, über den unser **Menu** gesetzt werden soll. (In unserem Fall also ein **Glyph** der Klasse **Linie**.)

Die Memberfunktion `Menu::CreateMenu` unterstützt den Konstruktor der Klasse bei dem Aufbau des **Menüs**, indem sie die **Menüleiste** samt **Pulldownmenüs** erzeugt und zusammenfügt.

Mit `TestMenu::compose` wird der Ergebnis – **Glyph** aus **Menü** und **Linie** – Glyph zusammengesetzt. `TestMenu::copyright` und `TestMenu::preferences` werden von den entsprechenden Menüeinträgen des erzeugten **Menüs** aufgerufen. Das Makro **declareActionCallback(TestMenu)** bewirkt die Deklaration der Struktur für einen **Callback** auf die Klasse **TestMenu**.

Die Implementation der Klassen **TestMenu** und **ActionCallback(TestMenu)** erfolgt in **TestMenu.c**:

```
#include "header.h"

implementActionCallback(TestMenu)

TestMenu::TestMenu(Glyph* screen_) {
       my_glyph = screen_;
}

PolyGlyph* TestMenu::compose(WidgetKit &kit) {
      const LayoutKit& layout = *LayoutKit::instance();
      Menu *m = CreateMenu(kit);
      return layout.vbox(m, kit.inset_frame(
             layout.margin(my_glyph, 10.0) )
                        );
}

Menu* TestMenu::CreateMenu(WidgetKit &kit) {
       Menu *file = kit.pulldown();
         MenuItem *m11 = kit.menu_item(
                  kit.fancy_label("Copyright"));
         m11->action(new ActionCallback(TestMenu)
                      (this,&TestMenu::copyright));
         file->append_item(m11);
```

```
        MenuItem *m12 = kit.menu_item(
                 kit.fancy_label("Quit"));
        m12->action(new ActionCallback(TestMenu)
                          (this,&TestMenu::quit));
        file->append_item(m12);

      Menu *options = kit.pulldown();
        MenuItem *m21 = kit.menu_item(
                 kit.fancy_label("Preferences"));
        m21->action(new ActionCallback(TestMenu)
                    (this,&TestMenu::preferences));
        options->append_item(m21);

     Menu *mb = kit.menubar();
       MenuItem *m1 = kit.menubar_item(
                kit.fancy_label("file"));
       MenuItem *m2 = kit.menubar_item(
                kit.fancy_label("options"));
       m1->menu(file);
       m2->menu(options);
       mb->append_item(m1);
       mb->append_item(m2);
    return mb;
}

void TestMenu::copyright() {
    cerr<<"TestMenu::copyright called\n";
}

void TestMenu::preferences() {
    cerr<<"TestMenu::preferences called\n";
}

void TestMenu::quit() {
    Session::instance()->quit();
}
```

Das Makro **implementActionCallback(TestMenu)** implementiert die Struktur für einen **Callback** auf die Klasse **TestMenu**. Die Memberroutine `TestMenu::compose` setzt die von „**CreateMenu**“ erzeugte **Menüleiste** und den vorgegebenen **Linie** – Glyph zusammen. Sie verwendet dabei eine vertikale **Box** des **LayoutKits**, und setzt den **Linie** – **Glyph** vorher noch in einen

Rahmen (**inset_frame**) und eine Umrandung (**margin**) der Breite **10.0**.

Die Routine `TestMenu::CreateMenu` leistet die eigentliche Konstruktion der gewünschten Menüstruktur. In der ersten Zeile wird ein **Pulldownmenü** „**file**“ mit Hilfe des **WidgetKit**–Objekts erzeugt. Danach werden die beiden Menüeinträge „**Copyright**“ und „**Quit**“ durch `WidgetKit::menu_item` erzeugt, der zugehörige **ActionCallback** erstellt und in das jeweilige **MenuItem** durch `MenuItem::action` eingetragen, sowie das **MenuItem** mit `Menu::append_item` an das **Pulldownmenü** angehängt. Dabei wird zur besseren Optik der `WidgetKit::fancy_label` als Beschriftung für den Menüeintrag verwendet. Der erzeugte **ActionCallback** zeigt jeweils auf eine Memberfunktion von **TestMenu**, die den denselben Namen wie das **MenuItem** trägt.

Gleichermaßen wird das Pulldownmennü „**Options**“ mit dem MenuItem „**Preferences**“ erzeugt und eingetragen.

Danach wird mittels `WidgetKit::menubar` eine Menüleiste erzeugt. Für jedes **Pulldownmenü** wird ein **MenuItem** mit gleichlautendem Namen erstellt, dem das **Pulldownmenü** durch `MenuItem::menu` übergeben wird. Schließlich werden die beiden **MenuItems**, die die **Pulldownmenüs** enthalten, in die Menüleiste mit `Menu::append_item` eingefügt.

Als Ergebnis liefert `TestMenu::CreateMenu` einen Zeiger auf die fertig zusammengesetzte Menüleiste.

Die zwei Memberfunktionen „**copyright**“ und „**preferences**“ werden beim Anklicken der entsprechenden Menüeinträge aufgerufen. Sie geben lediglich eine Kontrollmeldung auf der Standardfehlerausgabe aus. Eine sinnvollere Implementation bleibt Ihnen überlassen. Die Memberfunktion `TestMenu::quit` beendet das Programm über `Session::quit`. Hier wäre genauso `WidgetKit::quit` möglich. Das Hauptprogramm **main.c** sieht so aus:

```
#include "header.h"

int main(int argc, char **argv) {

   Session *session = new Session("Example",argc,argv);
   Style *style = session->style();
   WidgetKit& kit = *WidgetKit::instance();
   Linie *li = new Linie();
   Background *bg = new Background(li, kit.background());
   TestMenu* tm = new TestMenu(bg);

   Window *window = new ApplicationWindow(tm->compose(kit));

   window->place(100,100);
   return session->run_window(window);
}
```

Es unterscheidet sich von **main.c** aus Beispiel 3 nur dadurch, daß das **Background**–Objekt , das das **Linie** – Objekt enthält, nicht mehr direkt and das **ApplicationWindow** übergeben wird, sondern zuerst an das neue Objekt der Klasse **TestMenu**. Dieses kreiert ein **Menü** und fügt dies mit dem **Linie** – Glyph mittels `TestMenu::compose` zusammen. Der entstandene **PolyGlyph** wird dem **ApplicationWindow** zur Darstellung im Konstruktor übergeben. Als letzte Neuerung geben wir die Möglichkeit an, mittels `Window::place` das Fenster sofort beim Aufruf zu Positionieren.

Schreiben Sie nun ein geeignetes **Imakefile** und übersetzen Sie das Programm. Wenn Sie es auf einem Farbbildschirm oder einem Monochromschirm mit Farbemulation starten, wird das **Menü** im **Motif**–Erscheinungsbild zu sehen sein. Durch Aufruf des Programms mittels „<**Programmname**> **–openlook**“ erzeugt *InterViews* automatisch ein **WidgetKit**–Objekt, das **OpenLook**–Graphikelemente konstruiert. Wird das Programm auf einem monochromen Bildschirm gestartet oder geben Sie explizit „<**programmname**> **–monochrome**“ an, so erhält Ihre Applikation ein auf monochrome Bildschirme abgestimmtes Erscheinungsbild.

Kapitel 5

Dialog–Box mit Editoreingabe

In diesem Kapitel wollen wir die Implementation von einfachen Eingabefenstern besprechen, die wir für die Abfrage von speziellen Eingabedaten vom Benutzer zur Laufzeit des Programms auf dem Bildschirm erscheinen und nach Beendigung der Eingabe wieder verschwinden lassen können. Das Beispielprogramm 5 finden Sie ab Seite 81.

5.1 Aufgabenstellung

Als Beispiel wird eine **Dialogbox** aufgebaut, die in einem eigenen Fenster auf dem Bildschirm dargestellt werden soll. Die **Dialogbox** wird einen kleinen Stringeditor enthalten, mit dem der Benutzer seinen Namen eingeben kann. Der Aufruf der **Dialogbox** wird in das **Menü** von Kapitel 4 eingebettet. Das **Pulldownmenü „Options“** erhält zu diesem Zweck den zusätzlichen Menüeintrag „**DialogBox**“.

5.2 Die Window – Klassenhierarchie

Bereits in Abschnitt 1.3.9 wurden die Klassen für die Erzeugung von Bildschirmfenstern kurz erwähnt. Bisher wurde lediglich ein Fenster für jede Applikation geöffnet. Größere *InterViews* – Anwendungen machen aber häufig von mehreren Fenstern Gebrauch. Auch die **Dialogbox**, die in diesem Kapitel aufgebaut werden soll, wird in einem eigenen Fenster am Bildschirm dargestellt. Daher wollen wir uns an dieser Stelle eingehender mit dem Thema beschäftigen. Abbildung 5.1 zeigt die Ableitungsstruktur der Klassen für die Erzeugung von Bildschirmfenstern.

5.2.1 Window

Window ist die abstrakte Basisklasse aller dieser Klassen. Der für uns wichtige Teil der Klassendeklaration hat folgende Gestalt:

```
class Window {
  protected:
    Window(Glyph*);
  public:
    virtual void place(Coord left, Coord bottom);
    virtual void align(float x, float y);
    virtual void map();
    virtual void unmap();
    virtual void style(Style*);
    ...
};
```

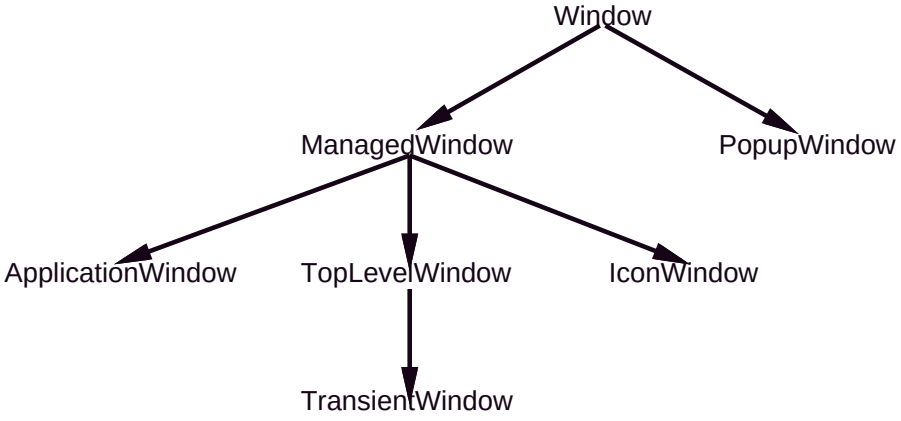


Abbildung 5.1: Ableitungsstruktur der Window–Klassen

Die Memberfunktion `Window::place` plaziert das Fenster so am Bildschirm, daß sein Ursprung mit den angegebenen Koordinaten zusammenfällt. Die Lage des Fensterursprungs wird durch `Window::align` festgelegt. Als Parameter erhält diese Routine zwei **float**–Werte zwischen **0.0** und **1.0**. Die Semantik dieser Parameter folgt denselben Regeln, wie sie für die Festlegung eines Glyphursprungs bereits ausführlich beschrieben wurden. Beispielsweise setzt das Argumentepaar (**0.5,0.5**) den Fensterursprung in die Mitte des Fensters. `Window::map` bildet das Fenster auf dem Bildschirm ab, während es mittels `Window::unmap` wieder vom

Bildschirm entfernt wird. Wichtig ist, daß die Objekte aller von **Window** abgeleiteter Klassen keine *shared objects* sind und mit *delete* gelöscht werden müssen.

Mit Hilfe von `Window::style` kann einem Fenster ein **Style**–Objekt übergeben werden, und damit jedes Attribut, das dieses **Style** enthält. Auf diese Art und Weise ist es z.B. möglich, dem **Window** einen Namen zu geben. Der Fenstername erscheint in der Titelleiste. Defaultwert ist hierbei der Name des Programms, das das Fenster erzeugt hat.

Von der Klasse **Window** sind gemäß Abbildung 5.1 zwei Klassen abgeleitet, die im folgenden beschrieben werden.

5.2.2 PopupWindow

Objekte dieser Klasse werden für Fenster verwendet, die nur eine sehr kurze Lebensdauer haben. Außerdem werden solche Fenster ohne Kenntnis des Windowmanagers abgebildet, d.h. sie erhalten keinen Rahmen und können auch nicht während des Programmlaufs vergrößert oder verkleinert werden. Unser Beispielprogramm verwendet solch ein **Window** als Dialogfenster. **PopupWindows** werden auch bei der Konstruktion von **Pulldown**- oder **Popupmenüs** verwendet. Die Klasse **WidgetKit** nimmt uns diese Konstruktionsarbeit weitgehend ab. Auch für unser **PopupWindow** im Beispiel 5, das ein Dialogfenster verwirklichen soll, benützen wir zum generieren die Hilfestellung der Klasse **Dialog** zum generieren, deswegen wird hier auf eine weitergehende Darstellung dieser Klasse verzichtet.

5.2.3 ManagedWindow

Die Klasse **ManagedWindow** bildet die abstrakte Basisklasse für alle Klassen, die Fenster für die Interaktion mit einem Windowmanager erzeugen. So dienen auch ihre Memberfunktionen zum größten Teil der Erledigung von Verwaltungsaufgaben. Einen kleinen Ausschnitt aus der großen Anzahl der verfügbaren Routinen zeigt der hier abgedruckte Teil der Klassendeklaration:

```
class ManagedWindow : public Window {
  protected:
    ManagedWindow(Glyph*);
  public:
    virtual void iconify();
    virtual void deiconify();
    ...
};
```

Sicherlich ist Ihnen bekannt, daß ein Fenster durch Anklicken eines bestimmten Feldes im Fensterrahmen als **Icon** dargestellt werden kann. Durch Anklicken dieses **Icons** wird das Fenster wieder in Normalgröße auf dem Bildschirm abgebildet. Dieselbe Funktionalität bieten die beiden Routinen `ManagedWindow::iconify` und `ManagedWindow::deiconify`. Der Name des **Icons** ist normalerweise derselbe, den das **Window** trägt, er kann aber durch Besetzen des **Style** – Attributes „**icon_name**“ im **Window** – **Style** geändert werden. Darüberhinaus sind unter anderen noch Memberfunktionen vorgesehen, durch die das Aussehen der **Icons** beeinflußt werden kann. Diese und weitere Routinen werden im Referenzmanual [MAN3.1] beschrieben.

5.2.4 ApplicationWindow

Das Hauptfenster einer *InterViews* – Applikation sollte von einem Objekt der Klasse **ApplicationWindow** erzeugt werden. Außerdem darf jede Anwendung nur ein Hauptfenster haben. Die Klasse **ApplicationWindow** stellt außer dem Konstruktor keine zusätzlichen Memberfunktionen bereit. Jedoch sind sämtliche von **Window** bzw. **ManagedWindow** ererbten Memberroutinen verfügbar. Die Deklaration sieht wie folgt aus:

```
class ApplicationWindow : public ManagedWindow {
  public:
    ApplicationWindow(Glyph*);
    ...
};
```

Der Konstruktor erhält als Parameter einen Zeiger auf die Wurzel einer Glyphhierarchie, die mindestens ein Element enthält. Diese Hierarchie wird bei der Abbildung des Fensters automatisch als Fensterinhalt am Bildschirm dargestellt.

5.2.5 TopLevelWindow

Alle anderen „normalen“ Fenster einer Applikation sind vom Typ **TopLevelWindow**. Die Deklaration der Klasse hat folgende Gestalt:

```
class TopLevelWindow : public ManagedWindow {
  public:
    TopLevelWindow(Glyph*);
    virtual void group_leader(Window*);
    virtual Window* group_leader() const;
    ...
};
```

Der Konstruktor erhält als Argument wiederum einen Zeiger auf die Wurzel einer Glyphhierarchie. Als Besonderheit besitzen **TopLevelWindows** einen sogenannten „**group_leader**“. Diese Rolle übernimmt im Regelfall das Hauptfenster der Anwendung, aber abhängig vom verwendeten Windowmanager können auch weitere Gruppierungen sinnvoll sein, was jedoch hier nicht weiter ausgeführt wird. Der „**group_leader**“ kann durch die beiden Memberfunktionen gesetzt bzw. zurückgeliefert werden. Manche Windowmanager erlauben es, daß bestimmte Operationen (z.B. Ikonifizieren), die auf dem „**group_leader**“ ausgeführt werden, zugleich auf alle Fenster der Gruppe angewandt werden.

5.2.6 TransientWindow

Objekte dieser von **TopLevelWindow** abgeleiteten Klasse werden für temporäre Fenster verwendet. Im Gegensatz zu **PopupWindows** werden **TransientWindows** aber vom Windowmanager verwaltet, haben also auch einen eigenen Namen und eine eigene Titelleiste. Allerdings werden sie von manchen Windowmanagern zur Unterscheidung von *normalen* Fenstern beispielsweise mit einem anderen Rahmen oder **Icon** gekennzeichnet. Der für uns interessante Deklarationsausschnitt sieht so aus:

```
class TransientWindow : public TopLevelWindow {
  public:
    TransientWindow(Glyph*);
    virtual void transient_for(Window*);
    virtual Window* transient_for() const;
    ...
};
```

Jedes **TransientWindow** sollte mit dem Fenster assoziiert werden, für das es temporäre Aufgaben, ausführt. Manche Windowmanager entfernen ein **TransientWindow** automatisch vom Bildschirm, wenn das assoziierte Fenster ikonifiziert wird. Mittels der beiden „**transient_for**“–Memberfunktionen wird das assoziierte Fenster gesetzt bzw. abgefragt.

5.2.7 IconWindow

Objekte dieser Klasse werden am Bildschirm abgebildet, wenn ihr assoziiertes Fenster ikonifiziert wird, d.h. sie ersetzen bzw. ergänzen die herkömmlichen **Icons**. Wir verwenden diese Klasse nicht, sie wird hier nur der Vollständigkeit halber erwähnt.

5.3 Weitere notwendige InterViews – Klassen

In diesem Abschnitt werden die Klassen **Button**, **FieldEditor**, **Dialog** und **DialogKit** neu eingeführt.

5.3.1 Button

Objekte dieser Klasse gehören zu den meistverwendeten Bausteinen graphischer Oberflächen, sie realisieren virtuell mechanische Knöpfe, mit denen durch Anklicken mit der Maus vielfältigste Aktionen gestartet oder Optionen ausgewählt werden können. **Buttons** werden z.B. eingesetzt, wenn in einer **Dialogbox** eine bestimmte Eingabe bestätigt („**OK–Button**“) oder zurückgewiesen („**Cancel–Button**“) werden soll.

Um eine möglichst einheitliche Benutzeroberfläche zu erhalten, sollten **Buttons** stets mit Hilfe eines **WidgetKit**–Objekts erzeugt werden. Aus diesem Grund wird die Klasse hier nicht näher vorgestellt. Wichtig ist nur, daß sie von **Glyph** abgeleitet ist. Damit können **Buttons** durch Einbau in eine Glyphhierarchie in eine Benutzeroberfläche eingefügt werden.

Die Erzeugung mittels **WidgetKit** ist recht einfach. Dazu schauen wir uns noch kurz ein paar **Button** – relevanten Memberfunkionen von **WidgetKit** an:

```
class WidgetKit {
  virtual Button* push_button(const char*, Action*) const;
  virtual Button* push_button(Glyph*, Action*) const;
  virtual Button* default_button(const char*, Action*) const;
  virtual Button* default_button(Glyph*, Action*) const;
  virtual Button* check_box(const char*, Action*) const;
  virtual Button* check_box(Glyph*, Action*) const;
  virtual Button* palette_button(const char*, Action*) const;
  virtual Button* palette_button(Glyph*, Action*) const;
  ...
};
```

Hierbei wird der Funktion, die den **Button** generiert, zusammen mit dem Aussehen des **Buttons** – das durch einen **Glyph** oder eine Zeichenkette dargestellt wird – die **Action** übergeben, die bei Anklicken des **Buttons** ausgeführt werden soll. Probieren Sie am besten einmal einige dieser **Buttons** aus; die visuelle Darstellung bietet Ihnen weitaus mehr, als ich hier beschreiben vermag.

5.3.2 FieldEditor

In einer **Dialogbox** werden in der Regel bestimmte Eingaben mit Hilfe von kleinen Texteditoren erledigt. Auch für diesen Zweck stellt *InterViews* Klassen zur Verfügung, z.B. die Klasse **FieldEditor**. Objekte dieser Klasse realisieren Editoren für eine Zeile Text. *InterViews* bietet mit dem **DialogKit** die Möglichkeit zur einfachen Erzeugung eines **FieldEditors** mit einem bestimmten generischen Erscheinungsbild mit Hilfe von **WidgetKit**–Objekten. **FieldEditor** ist von **InputHandler** und somit auch von**MonoGlyph** abgeleitet. Daher kann ein Objekt dieser Klasse in eine Glyphhierarchie und damit in eine Benutzeroberfläche eingefügt werden. Da diese Klasse von **InputHandler** abgeleitet ist, können Sie natürlich auch auf „**keystrokes**“ oder „**mausklicks**“ im Inneren des **FieldEditors** gesondert reagieren, indem Sie die virtuellen **InputHandler**–Funktionen ausprogrammieren; so ist z.B. eine Auswahl von eingebbaren Buchstaben während des editierens möglich.

Bevor wir uns den Memberfunktionen zuwenden, soll zunächst die Bedienung eines **FieldEditors** erklärt werden. Folgende wichtigen Tastenkombinationen steuern den Editiervorgang:

```
<ctrl>-b:     ein Zeichen nach links (backspace)
<ctrl>-f:     ein Zeichen nach rechts (forward)
<ctrl>-a:     an den Anfang der Zeile springen
<ctrl>-e:     an das Ende der Zeile springen
<ctrl>-d:     das Zeichen unter dem Cursor loeschen
              (delete)
<Backspace>:  das Zeichen links vom Cursor loeschen
```

Während Sie die mittlere Maustaste gedrückt halten, können Sie innerhalb des Editors bei größeren Texten nach rechts und links scrollen.

Wenden wir uns nun einem Deklarationsausschnitt der Klasse **FieldEditor** zu:

```
class FieldEditor : public InputHandler {
  public:
    FieldEditor(const String& sample, WidgetKit*, Style*,
                FieldEditorAction* = nil);
    virtual const String* text() const;
    ...
};
```

Aus den verfügbaren Memberfunktionen der Klasse **FieldEditor** habe ich die beiden für uns wichtigsten ausgewählt.

Der Konstruktor erhält bis zu vier Parameter. Das erste Argument wird mit einem Beispielstring besetzt, der für die Berechnung der Größe des Editors verwendet wird und bei der Abbildung des Editors am Bildschirm in der Editierfensterzeile erscheint. Der FieldEditor stützt sich hier im Gegensatz zu dem aus Version 3.0 auf ein Objekt der *InterViews* – Klasse **String**.

Die als Zeiger übergebenen Objekte **WidgetKit** und **Style** regeln das Aussehen des Editors. Das optional anzugebende **FieldEditorAction**–Objekt wird aktiviert, wenn der Editiervorgang durch Drücken der <RETURN>–Taste regulär beendet wird. Diese **Action** kann mit einem normalen **ActionCallback** per Standard – Klassenkonvertierung besetzt werden; wie, können Sie anhand unseres Beispiels später sehen. Die **FieldEditorAction** ist mit „**nil**“ vorbelegt, d.h. wenn hier nichts beim initialisieren der Klasse übergeben wird, reagiert der **FieldEditor** auch nicht gesondert auf das Drücken von <RETURN>. Der eingegebene Text kann durch die Memberfunktion `FieldEditor::text` abgefragt

werden, sie liefert ein **String**–Objekt zurück. In diesem kann die enthaltene Zeichenkette durch die Memberfunktion `String::string` abgefragt werden. Falls die Abfrage während des Editiervorgangs stattfindet, bitte ich zu beachten, daß der Text–String durch weitere Eingaben verändert wird. Kopieren Sie in diesem Fall am besten den **String** in ein neues **String**–Objekt, bevor Sie damit weiterarbeiten.

5.3.3 DialogKit

DialogKit ist ein Tool zur einfachen Erstellung von Dialogfenstern höherer Ordnung. In zukünftigen *InterViews* – Versionen sind hier noch einige zusätzliche Funktionen vorgesehen, im Moment finden wir hier nur die folgenden:

```
class DialogKit {
  protected:
    DialogKit();
  public:
    static DialogKit* instance();
    virtual FieldEditor* field_editor(
      const char* sample, Style*, FieldEditorAction* = nil
    ) const;
    virtual FileChooser* file_chooser(
      const char* dir, Style*, FileChooserAction* = nil
    ) const;
    ...
};
```

Wie Die anderen *Kit's* wird auch **DialogKit** mit `DialogKit::instance` verwaltet und pro **Session** nur einmal erzeugt.
Die Memberfunktion `DialogKit::FieldEditor` generiert einen **FieldEditor** mit den übergebenen Parametern. Der Konstruktor deckt sich mit dem der Klasse **FieldEditor**, mit Ausnahme des **WidgetKits**, das hier nicht mehr extra übergeben werden muß. `DialogKit::FileChooser` erzeugt ein Auswahlfenster, das die Auswahl eines Files aus einer Liste von Files in dem angegebenen Verzeichnis „**dir**“ ermöglicht.

5.3.4 Dialog

Die Klasse **Dialog** stellt uns ein einfaches **PopupWindow** zur Verfügung, das wir bei Bedarf auf dem **Display** öffnen können:

```
class Dialog : public InputHandler {
  public:
    Dialog(Glyph*, Style*);
    boolean post_for(Window*);
    boolean post_at(Coord x, Coord y);
    virtual boolean run();
    virtual void dismiss(boolean accept);
    ...
};
```

Dialog ist von **InputHandler** abgeleitet. Der Konstruktor erwartet die Übergabe eines Zeigers auf ein **Style**–Objekt und auf den **Glyph**, den das Fenster darstellen soll. Mit der Memberfunktion `Dialog::post_for` wird das Dialogfenster mit dem enthaltenen **Glyph** in der Mitte des angegebenen **Window** dargestellt. Mit `Dialog::post_at` kann auch eine diskrete Position auf dem **Display**, auf der das Fenster dargestellt werden soll, festgelegt werden. Durch die Memberfunktion `Dialog::run` wird eine Warteschleife gestartet, die wartet, bis das Dialogfenster mit `Dialog::dismiss` geschlossen wird. Mit der Funktion `Dialog::dismiss` wird das Dialogfenster wieder entfernt; der in der Funktion übergebene boolsche Parameter „**accept**“ wird von den Memberfunktionen `Dialog::post_for`, `Dialog::post_at` und `Dialog::run` zurückgeliefert, und kann Aufschluß über die Umstände geben, unter denen das Fenster geschlossen wurde, je nachdem, was in `Dialog::dismiss` übergeben wird.

5.3.5 String

InterViews besitzt eine eigene Klasse für Stringoperationen. Sie soll hier kurz vorgestellt werden, da sich einige Memberfunktionen nur durch **String**–Objekte bedienen lassen, so z.B. `FieldEditor::text`.

```
class String {
  public:
    String();
    String(const char*);
    const char* string() const;
    int length() const;
    virtual String substr(int start, int length) const;
    virtual int search(int start, u_char) const;
    ...
};
```

Ausser den oben angegebenen Memberfunktionen gibt es noch einige weitere Funktionen, so z.B. Vergleichs- oder Convertierungsoperatoren für **String**–Objekte. Wir wollen hier jedoch nicht weiter darauf eingehen. Der Leser möge bei Interesse im Referenzmanual [MAN3.1] oder in der zugehörigen **include**–Datei <**OS/string.h**> in der *InterViews* – Installation nachsehen.

Der Konstruktor kann ohne Argument oder mit einer Zeichenkette („**char***“) aufgerufen werden. Die Memberfunktion `String::string` liefert diese Zeichenkette zurück. Hierbei verdient Beachtung, daß die Zeichenkette als Konstante übergeben, gespeichert und zurückgeliefert wird. `String::length` ergibt die Länge der Zeichenkette, und `String::substr` den von ab Position „**start**“ „**length**“ Zeichen langen Teil davon. `String::search` durchsucht die Zeichenkette ab Position „**start**“ nach Auftreten des Zeichens „**u_char**“; falls das Zeichen gefunden wird, wird die Position in der Zeichenkette zurückgeliefert, ansonsten „**0**“.

5.4 Beispiel 5

Nach diesen umfangreichen Erklärungen, in denen Sie einiges über die *InterViews* – Konzepte gelernt haben, wollen wir nun an die Implementierung der eingangs erwähnten Aufgabenstellung gehen. Wenn Sie dieses Programm verstehen, sind Sie bereits in der Lage, größere *InterViews* – Anwendungen selbst zu schreiben.

Der Sourcecode ist auf die Files **header.h**, **linie.h**, **linie.c**, **selector.h**, **selector.c**, **testmenu.h**, **testmenu.c**, **mydialog.h**, **mydialog.c** und **main.c** ver-

teilt. (*Die Klasse Selector wurde nur der Zusammenfassung halber wieder eingesetzt, sie wurde gegenüber Kapitel 3 nicht verändert.*) Die Datei **header.h** wurde aus Beispiel 3 und 4 zusammengesetzt, und um die *InterViews* – Includedateien für **Dialog**, **DialogKit** und **String** erweitert. Außerdem wurde das File **mydialog.h** einbezogen, das die Deklaration unserer neuen Klasse **MyDialog** (für die Erzeugung der **Dialogbox**) enthält:

```
#ifndef header_h
#define header_h

#include <IV-look/kit.h>
#include <IV-look/dialogs.h>

#include <InterViews/session.h>
#include <InterViews/window.h>
#include <InterViews/display.h>
#include <InterViews/event.h>
#include <InterViews/style.h>
#include <InterViews/background.h>
#include <InterViews/color.h>
#include <InterViews/brush.h>

#include <InterViews/dialog.h>
#include <InterViews/layout.h>
#include <OS/string.h>
#include <stream.h>

#include "linie.h"
#include "selector.h"
#include "mydialog.h"
#include "testmenu.h"

#endif
```

Die beiden Dateien **linie.h** und **linie.c** sind gegenüber Kapitel 4 unverändert und können übernommen werden, genauso wie **selector.h** und **selector.c** aus Kapitel 3. Dagegen haben **testmenu.h** und **testmenu.c** im Vergleich zu Kapitel 4 kleine Änderungen erfahren, die im wesentlichen mit der Erzeugung und Löschung der **Dialogbox** zusammenhängen.

testmenu.h hat nun folgende Gestalt:

```
#ifndef testmenu_h
#define testmenu_h

class TestMenu {
public:
        TestMenu(Glyph*);
        PolyGlyph* compose(WidgetKit&);
        Menu* CreateMenu(WidgetKit&);

        void quit();
        void copyright();
        void preferences();
        void call_dialogbox();

        void set_window(ApplicationWindow*);
        void delete_dialogbox();
private:
        ApplicationWindow *menuwindow;
        MyDialog *MD;
        Glyph* my_glyph;
};

declareActionCallback(TestMenu)

#endif
```

Die neue Memberfunktion `TestMenu::set_window` ist dazu da, einen Zeiger auf das Hauptfenster unserer Applikation, das **ApplicationWindow** zu übergeben. Dieser Zeiger wird dann in der Variablen „**menuwindow**“ abgespeichert, und später dazu verwandt, die **Dialogbox** in der Mitte dieses Fensters darzustellen. `TestMenu::set_window` wird aus unserem Hauptprogramm **main.c** aufgerufen, nachdem das **ApplikationWindow** mit unserem **TestMenu** als Bestandteil erzeugt wurde.

Die Memberfunktionen `TestMenu::call_dialogbox` und `TestMenu::delete_dialogbox` sind für das Erzeugen und Löschen der **Dialogbox** zuständig. Neben dem oben erwähnten „**menuwindow**“ wird noch eine weitere private Membervariable „**MD**“ eingeführt, die einen Zeiger auf die zu erzeugende **Dialogbox** enthalten soll.

Das gegenüber Kapitel 4 ebenfalls veränderte File **testmenu.c** hat nun fol-

gendes Aussehen:

```
#include "header.h"

implementActionCallback(TestMenu)

TestMenu::TestMenu(Glyph* screen_) {
        my_glyph = screen_;
        menuwindow = nil;
        MD = nil;
}

PolyGlyph* TestMenu::compose(WidgetKit &kit) {
        const LayoutKit& layout = *LayoutKit::instance();
        Menu *m = CreateMenu(kit);
        return layout.vbox(m, kit.inset_frame(
               layout.margin(my_glyph, 10.0) )
                         );
}

Menu* TestMenu::CreateMenu(WidgetKit& kit) {
       Menu *file = kit.pulldown();
         MenuItem* m1 = kit.menu_item(
                        kit.fancy_label("Copyright") );
               Action *copyright = new ActionCallback(TestMenu)
                                   (this,&TestMenu::copyright);
               m1->action(copyright);
         MenuItem* m2 = kit.menu_item(
                        kit.fancy_label("Quit") );
               Action *quit = new ActionCallback(TestMenu)
                                   (this,&TestMenu::quit);
               m2->action(quit);
         file->append_item(m1);
         file->append_item(m2);

       Menu *options = kit.pulldown();
         MenuItem* m3 = kit.menu_item(
                        kit.fancy_label("Preferences") );
         Action *preferences = new ActionCallback(TestMenu)
                              (this,&TestMenu::preferences);
               m3->action(preferences);
         MenuItem* m4 = kit.menu_item(
```

```
                        kit.fancy_label("DialogBox") );
          Action *dialogbox = new ActionCallback(TestMenu)
                         (this,&TestMenu::call_dialogbox);
                m4->action(dialogbox);
        options->append_item(m3);
        options->append_item(m4);

        Menu *mb = kit.menubar();
          MenuItem *mb1 = kit.menubar_item(
                         kit.fancy_label("File") );
                mb1->menu(file);
          MenuItem *mb2 = kit.menubar_item(
                         kit.fancy_label("Options") );
                mb2->menu(options);
                mb->append_item(mb1);
                mb->append_item(mb2);
        return mb;
}

void TestMenu::set_window(ApplicationWindow *aw) {
        menuwindow = aw;
}

void TestMenu::copyright() {
        cerr<<"TestMenu::copyright called\n";
}

void TestMenu::preferences() {
        cerr<<"TestMenu::preferences called\n";
}

void TestMenu::quit() {
        Session::instance()->quit();
}

void TestMenu::call_dialogbox() {
        if ( MD != nil ) {
         cerr<<"Dialogbox already called\n";
         return;
        }
        WidgetKit& kit = *WidgetKit::instance();
        Action *pushaction =
                new ActionCallback(TestMenu)
```

```
                    (this,&TestMenu::delete_dialogbox);
        MD = new MyDialog(pushaction, kit.style());
        MD->post_for(menuwindow);
}

void TestMenu::delete_dialogbox() {
        const char* text;
        text = MD->get_editor()->text()->string();

        cerr<<"you entered: "<<text<<"\n";
        MD->dismiss(1);
        MD = nil;
}
```

Der Konstruktor besetzt nun zusätzlich die neueingeführten Membervariablen „**menuwindow**“ und „**MD**“ mit „**nil**“. In „**CreateMenu**“ ist die Zeile neu, in der die **Action** „**dialogbox**“ erzeugt wird. Die Ausführung dieser **Action** führt zum Aufruf der Routine „**call_dialogbox**“.

Außerdem wird mittels **options–>add_item** der Eintrag „**DialogBox**“ zusammen mit der zugehörigen **Action** „**dialogbox**“ in das **Pulldownmenü** „**options**“ eingefügt.

Die beiden Routinen „**copyright**“ und „**preferences**“ bleiben unverändert. Wichtig ist aber die neue Routine „**call_dialogbox**“. Sie wird automatisch aufgerufen, wenn der Menüeintrag „**DialogBox**“ aktiviert wird. Dies wird von der **Action** „**dialogbox**“ bewirkt.

In `TestMenu::call_dialogbox` wird zunächst überprüft, ob der Zeiger auf das Dialogfenster gleich „**nil**“ ist. Falls dies nicht gilt, wurde diese Funktion inzwischen schon einmal aufgerufen und dabei eine **Dialogbox** mit zugehörigem Fenster erzeugt, die aber noch nicht gelöscht wurde und sich daher noch auf dem Bildschirm befindet. In diesem Fall wird eine entsprechende Warnmeldung ausgegeben und die Ausführung der Routine beendet. Ansonsten werden nun die **Dialogbox** neu erzeugt:

Dazu wird zunächst ein Zeiger auf das **WidgetKit**–Objekt erstellt. Danach wird die **Action** „**pushaction**“ erzeugt, die bei der Aktivierung des **OK–Buttons** der **Dialogbox** ausgeführt werden soll. Mit den Parametern „**pushaction**“ und dem Standard–**Style** des **WidgetKits** wird nun im Konstruktor der Klasse **MyDialog** die **Dialogbox** erzeugt. Der Zeiger auf dieses Objekt wird in der Membervariablen „**MD**“ festgehalten.

Durch Aufrufen der Memberfunktion `Dialog::post_for` mit dem Parameter „**menuwindow**“ wird die **Dialogbox** in der Mitte dieses **Windows** dargestellt. Der WindowManager wartet hierbei solange, bis die Bearbeitung in der **Dialogbox** beendet ist, bis er nach `Dialog::post_for` weiterarbeitet.

Wird in der **Dialogbox** die **Action** „**pushaction**“ aufgerufen, so wird die Routine `TestMenu::delete_dialogbox` aktiviert. Dort wird zunächst der editierte Text auf der Standardfehlerausgabe ausgegeben. Dieser Text befindet sich in einem **String**–Objekt, das mit der Funktion `FieldEditor::text` abgefragt wird. Die Zeichenkette des **String**–Objekts erhalten wir durch `String::string`. Danach wird das Fenster durch `Dialog::dismiss` vom Bildschirm entfernt und das zugehörige Objekt gelöscht. Die Membervariable „**MD**“ wird schließlich noch explizit auf „**nil**“ gesetzt.

Sicher werden Sie sich jetzt fragen, weshalb weder der **FieldEditor** *noch das* **MyDialog**–*Objekt selbst explizit mit delete gelöscht werden. Dies liegt daran, daß beide Klassen* **Glyph** *und somit* **Resource** *als Basisklasse haben. Damit gehören sie zu den shared objects, die ja niemals durch delete gelöscht werden dürfen. Sie wissen sicherlich noch, daß shared objects mit* **ref–** *und* **unref–** *Aufrufen verwaltet werden und nach entsprechend vielen* **unref–***Aufrufen automatisch gelöscht werden.*

Damit ist die Diskussion der Klasse **TestMenu** abgeschlossen. Im folgenden wenden wir uns der neuen Klasse **MyDialog** zu. Die Deklaration der Klasse ist in der Datei **mydialog.h** zu finden:

```
#ifndef mydialog_h
#define mydialog_h

class MyDialog : public Dialog {
public:
        MyDialog(Action*, Style*, Glyph* = nil);
        ~MyDialog();
        FieldEditor* get_editor();
private:
        MonoGlyph* CreateGlyph(WidgetKit&);

        Action *pushaction;
        FieldEditor *editor;
};

#endif
```

Die Klasse **MyDialog** wird dazu verwendet, eine **Dialogbox** zu konstruieren. **MyDialog** ist von der Basisklasse **Dialog**, und somit auch von **InputHandler** und **MonoGlyph** abgeleitet. Die Basisklasse verlangt normalerweise einen **Style** mitsamt dem darzustellenden **Glyph** schon bei der Initialisierung. Dies werden wir umgehen, da wir den **Glyph** innerhalb unserer **Dialog**–Klasse er-

zeugen wollen, um unserer Direktive der Objektgebundenen Funktionen treu zu bleiben.

Die Klasse besitzt zwei private Membervariablen, und zwar die **Action „pushaction“**, die im Konstruktor übergeben wird, und die dazu dient, die **Dialogbox** wieder vom Bildschirm zu entfernen, und den FieldEditor „**editor**“, der selbst erzeugt wird, und zur Eingabe einer Zeile Text dient. Dieser kann von außen mit der Funktion `MyDialog::get_editor` abgefragt werden, um z.B. den eingegebenen Text am ende zu übergeben.

Der **Glyph** der **Dialogbox** besteht aus einer Zusammensetzung von **PolyGlyphs**, **Glues** und anderen Objekten des **LayoutKits** (siehe Abbildung 5.2). Er wird in der Memberfunktion `MyDialog::CreateGlyph` erzeugt.

Die Implementation der Klasse **MyDialog** ist in **mydialog.c** enthalten:

```
#include "header.h"

MyDialog::MyDialog(Action *act, Style *s, Glyph *g)
                                                : Dialog(g, s) {
        WidgetKit& kit = *WidgetKit::instance();
        DialogKit& dkit = *DialogKit::instance();
        pushaction = act;
        Resource::ref(pushaction);
        editor = dkit.field_editor("Beeblebrox", kit.style(),
                 (FieldEditorAction*)pushaction);
        Resource::ref(editor);
        body(CreateGlyph(kit));
        append_input_handler(editor);
        next_focus();
}

MyDialog::~MyDialog() {
        Resource::unref(editor);
        Resource::unref(pushaction);
}

FieldEditor* MyDialog::get_editor() {
        return editor;
}

MonoGlyph* MyDialog::CreateGlyph(WidgetKit &kit) {
        LayoutKit& l = *LayoutKit::instance();

        Glyph *v1 = l.vglue(25.0,40.0,10.0);
```

```
        Glyph *v2 = l.vglue(15.0,30.0,5.0);
        Glyph *h1 = l.hglue(25.0,100.0,10.0);

        PolyGlyph *label_box =
          l.hbox(h1, kit.outset_frame(
                     kit.fancy_label("Enter Name:")), h1);

        PolyGlyph *enter_box =
          l.hbox(h1, kit.outset_frame(editor), h1);

        PolyGlyph *exit_button =
          l.hbox(h1, kit.push_button("OK",pushaction), h1);

        MonoGlyph *my_all = kit.outset_frame(
          l.vbox(v1, label_box, v2, enter_box,
                                v2,exit_button,v1));

        return my_all;
}
```

Im Konstruktor der Klasse **MyDialog** lassen wir uns die **Action** zur Beendigung der **Dialogbox**, einen Standard–**Style** und einen leeren Dummy–**Glyph** als *Body* übergeben. Wir bedienen unsere Basisklasse mit dem **Style** und dem Dummy–**Glyph**. Unseren *Body*–**Glyph** setzen wir nachher selbständig mit unserer vererbten Memberfunktion `MonoGlyph::body`. Dazu erzeugen wir eine **instance** des **WidgetKits** und des **DialogKits**. Die übergebene **Action** geben wir an unsere Memebervariable „**pushaction**“ weiter. Wir erzeugen unseren Texteditor durch das **DialogKit** mit `DialogKit::FieldEditor`. Dabei übergeben wir die Zeichenkette „Beeblebrox“, um die Größe festzulegen und das Standard–**Style** von **WidgetKit**. Als dritten Parameter übergeben wir unsere „**pushaction**“ an den Editor, um bei Drücken von <RETURN> denselben Effekt zu bekommen, wie wenn der „**OK**“–**Button** gedrückt wird. Hierbei müssen wir unsere normale **Action** aber zuerst in eine **FieldEditorAction** konvertieren, die vom Konstruktor von **FieldEditor** erwartet wird. Wir referenzieren witerhin „**pushaction**“ und „**editor**“. Schließlich erzeugen wir mit der Memberfunktion `MyDialog::CreateGlyph` die Darstellung der **Dialogbox** und setzen sie mit `MonoGlyph::body` als unseren *Body* fest. Um Eingaben in unseren Texteditor zu erleichtern, müssen wir diesen **InputHandler** (*Sie erinnern sich ?* **FieldEditor** *und* **Dialog** *sind von* **InputHandler** *abgeleitet.*) noch dem **InputHandler** von **MyDialog** mit
`InputHandler::append_input_handler` bekannt machen. Danach setzen wir auch gleich den sensitiven Eingabebereich (= focus) mit

`InputHandler::next_focus` auf den Texteditor. Dies hat zur Folge, daß wir, wenn die **Dialogbox** auf dem Bildschirm dargestellt wird, in unserem Texteditorfenster edititeren können, sobald der Mauszeiger sich innerhalb der Umrandung der **Dialogbox** befindet. Compilieren Sie das Programm einmal ohne die letzten zwei Zeilen, dann müssen Sie den Mauszeiger zum editieren auf jeden Fall im Bereich des Texteditors halten, was manchmal recht umständlich ist.

Um die Zusammensetzung der **Dialogbox** in `MyDialog::CreateGlyph` besser verstehen zu können, sehen wir uns zunächst die graphische Darstellung der realisierten Glyphhierarchie in Abbildung 5.2 an. Aus dieser Abbildung wird ersicht-

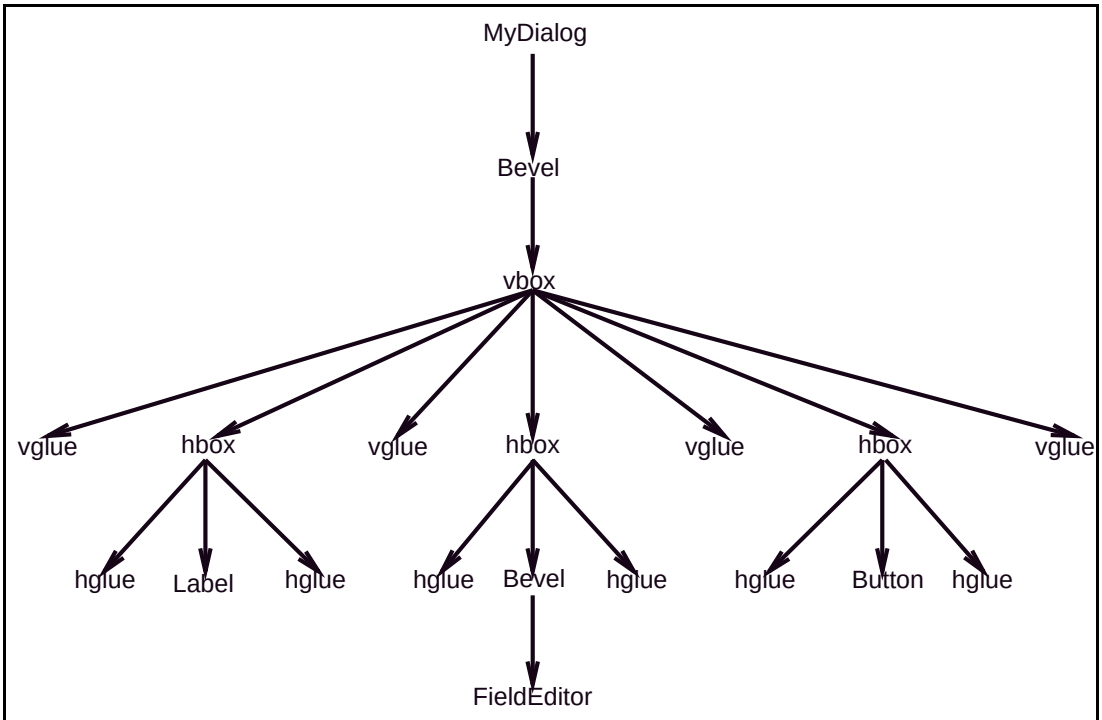


Abbildung 5.2: Struktur der Dialogbox

lich, daß die **Dialogbox** im wesentlichen aus einem **Label**, einem **FieldEditor** und einem **Button** besteht. Die übrigen Objekte sind nur für die Layoutgestaltung erforderlich. In Abbildung 5.3 ist die Struktur der erzeugten **Dialogbox** stilisiert dargestellt, wobei die einzelnen Komponenten zu Ihrer Orientierung beschriftet wurden. Die dargestellten Objektbegrenzungen werden selbstverständlich nicht auf dem Bildschirm abgebildet.

Beachten Sie, daß neben Objekten, die innerhalb einer **Box** zentriert werden sollen, **Glues** mit den gleichen charakteristischen Größen angegeben werden. Als Beispiel wird die `LayoutKit::hbox` betrachtet, die den **Button** enthält. Links und rechts des entsprechenden **Button**–Objekts befinden sich zwei `LayoutKit::hglues` mit denselben Parametern. Hätte der linke **Glue** die gleiche

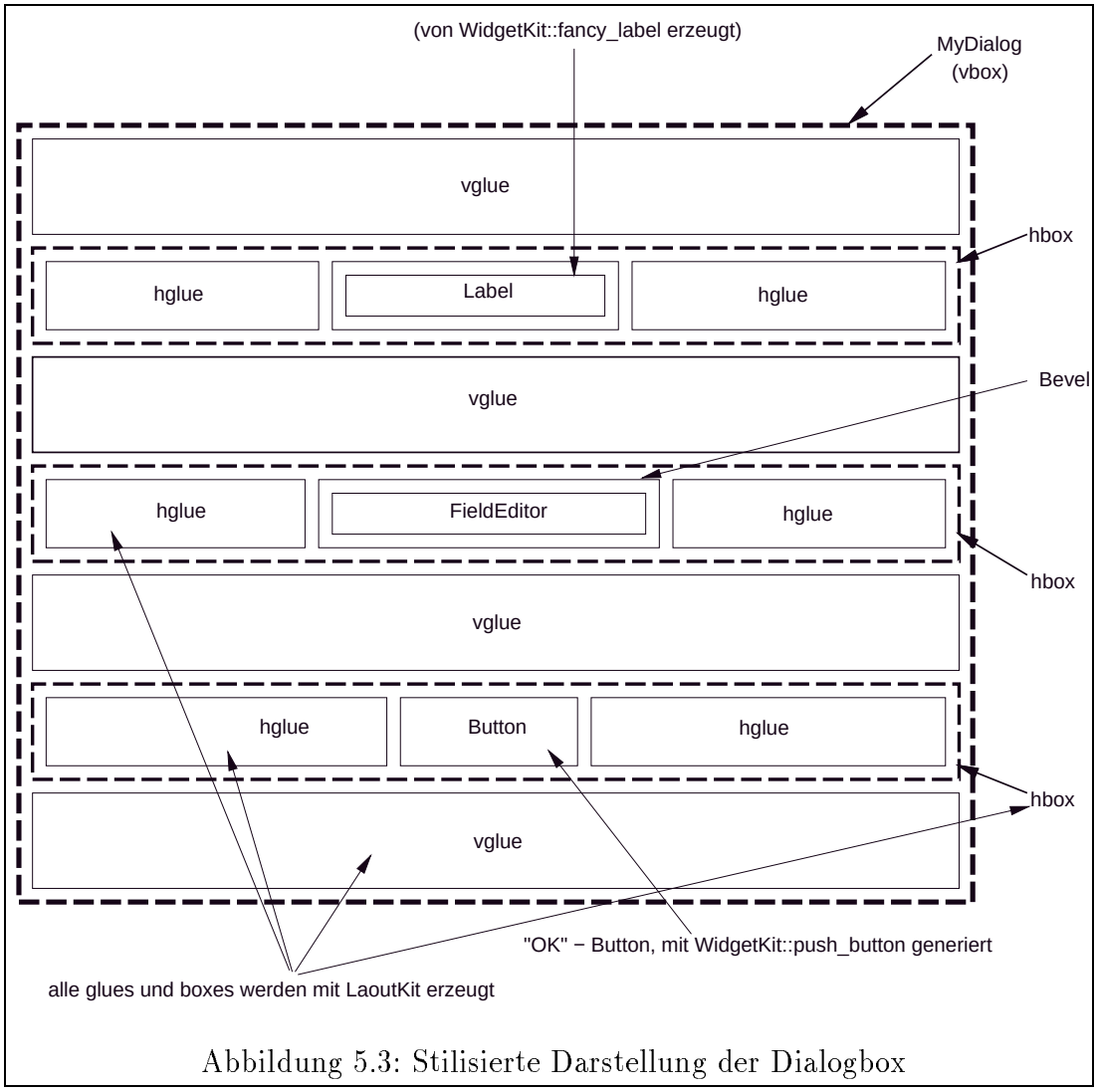


Abbildung 5.3: Stilisierte Darstellung der Dialogbox

natürliche Größe, jedoch eine kleinere Dehnbarkeit als der rechte, so würde in dem Fall, daß für die umfassende „**vbox**“ die Summe der natürlichen Größen der beiden Glues nicht mehr ausreicht, der rechte Glue stärker gedehnt. Damit würde sich der **Button** nach links verschieben. Denselben Effekt hätten Sie, wenn sie den **Button** in ein `LayoutKit::hcenter`–Objekt packen würden.

Hier werden die Einzelkomponenten in der horizontalen mit `LayoutKit::hbox` zusammengefaßt und schließlich mit `LayoutKit::vbox` in der vertikalen zusammengefügt. Um den Gesamt–**Glyph** legen wir noch einen Rahmen vom Typ `WigetKit::outset_frame`; hier dient dieser jedoch nicht nur zur Verschönerung, sondern ist sogar notwendig, weil ohne ihn der Inhalt des **Dialogbox**–Hintergrunds verloren ginge. (Es wird dann ein wirres Speichermuster dargestellt.) Aus welchen Gründen, ist mir nicht ersichtlich; wahrscheinlich ist dies ein Fehler in der noch nicht ganz vollständigen Implemetation der Klasse **Dialog**.

Schließlich fehlt noch die Beschreibung des Hauptprogramms, das in **main.c** enthalten ist:

```
#include "header.h"

int main(int argc, char **argv) {

    Session *session = new Session("Example",argc,argv);
    Style *style = session->style();
    WidgetKit& kit = *WidgetKit::instance();

    Linie *li = new Linie();
    Selector *handy = new Selector(li, style);
    Background *bg = new Background(handy, kit.background());

    TestMenu *tm = new TestMenu(bg);

    ApplicationWindow *window =
                      new ApplicationWindow(tm->compose(kit));
    tm->set_window(window);

    Display *display = session->default_display();
    window->align(0.5,0.5);
    window->place(display->width()/2,display->height()/2);
    style->attribute("name","Example 5");
    window->style(style);

    return session->run_window(window);
}
```

Im Prinzip laufen hier die gleichen Schritte wie in **main.c** aus Kapitel 4 ab. Zusätzlich wird noch die Funktion `TestMenu::set_window` aufgerufen, um das erzeugte **ApplicationWindow** anschließend gleich der Unterklasse **TestMenu** bekannt zu machen. Die Plazierung des **ApplicationWindow** erfolgt diesmal genau in der Mitte des **Displays**. Dem **Window** wird außerdem der **Style** mit dem Namen „Example 5“ übergeben. Dies soll Ihnen jedoch nur als Anregung dienen; so wird die Titleleiste des **ApplicatinWindows** mit einem Namen belegt, wie dies schon in Kapitel 3 angemerkt wurde.

Abschließende Bemerkungen

Wie Sie gesehen haben, bietet *InterViews* eine Fülle an Möglichkeiten der Gestaltung von graphischen Benutzeroberflächen. Dabei haben Sie bisher vielleicht die Hälfte der Klassen dieser Bibliothek – zumeist nur in Grundzügen – kennengelernt, und haben noch ein gutes Stück Arbeit vor sich, wenn Sie zu einem perfektem Konstrukteur von Bedieneroberflächen in *InterViews*, von der Qualität, wie ich sie im Vorwort erwähnt habe, werden wollen. Dieses Einführungsheft sollte Ihnen nur die Grundkenntnisse vermitteln, wie Sie an *InterViews* – Objekte herangehen können, und einfache Beispiele bieten, um mit den wichtigsten Klassen und Funktionen zurecht zu kommen. Die meisten „Features“ müssen Sie durch selbständiges Ausprobieren in eigenen Programmen – wozu Sie ja auch meine Beispiele erweitern können – kennenlernen.

Mein Programmierstil stellt natürlich nur einen Vorschlag dar. Mit Ausnahme der Stellen, an denen ich gute Gründe für meine Art, ein Programm anzugehen (wie z.B. Objektorientiertheit) gegeben habe, gibt es sicher mehrere Implementationen, die zu den selben, vielleicht sogar besseren Ergebnissen führen.

Ich hoffe, ich konnte Ihnen einige Hilfen und Anregungen beim Verständnis von *InterViews* geben, und Sie finden sich nun zumindest halbwegs im Referenzmanual [MAN3.1] zurecht. (Dieses ist gar nicht mehr so schwer zu verstehen, wenn man die wichtigsten Begriffe und Strukturen einmal verstanden hat.) Scheuen Sie sich auch nicht, in der Newsgroup „comp.windows.interviews“ einmal um Rat zu fragen; Sie werden hier auch bei Anfängerfragen freundliche Menschen finden, die Ihnen weiterhelfen, wenn Sie ein Problem haben. Der Grund liegt wahrscheinlich darin, daß Sie nicht der Einzige sind, der mit *InterViews* so seine Probleme hat, weil die Dokumentation einfach zu dürftig ist; dies soll die Mächtigkeit und Qualität von *InterViews* jedoch auf keinen Fall schmälern – im Gegenteil: Indem Sie diese Einführung in Händen halten, können Sie erkennen, daß *InterViews* für mich immerhin beeindruckend genug war, ein halbes Jahr an Arbeit darauf zu verwenden, anderen diese beispielhaft objektorientierte und portierbare Programmbibliothek näher zu bringen.

Ich hoffe zumindest, daß mit der nächsten Version von *InterViews*, sofern sie kommen sollte, dieses Manual nicht nutzlos wird, weil Strukturen wieder umgestellt werden. Vielleicht findet sich ja auch bei den für *InterViews* zuständigen Programmieren an der „Leland Stanford Junior University“ jemand, der ein Hi-

storyfile erstellt, das dann die Unterschiede von der alten zur neuen Version auflistet; damit wäre vielen schon geholfen. Die eine Seite am Anfang des „Reference Manuals“ zu Version 3.1 [MAN3.1] halte ich nicht für geeignet, als große Hilfe zu dienen.

Wie ich schon erwähnte, bezog ich meine Kenntnisse hauptsächlich aus der Arbeit von Stefan Mayer [IV3.0] und dem Reference Manual [MAN3.1]. Falls ich von Ihnen erkannte Fehler gemacht habe, so bitte ich Sie sogar darum, mir diese mitzuteilen, damit ich sie korrigieren kann, und nicht mit meiner Unwissenheit leben muß :).

Literaturverzeichnis

[MAN3.0] Mark A. Linton et al. *„InterViews Reference Manual Version 3.0“*. Board of Trustees of the Leland Stanford Junior University, 1991.

[MAN3.1] Mark A. Linton, Paul R. Calder, John A. Interrante, Steven Tang, John Vlissides. *„InterViews Reference Manual Version 3.1“*. Board of Trustees of the Leland Stanford Junior University, 1992.

[IV3.0] Stefan Mayer. *„Einführung in InterViews“*. Technische Universität München, 1992.

[PI] Walter E. Proebster. *„Peripherie von Informationssystemen“*. Springer–Verlag, Berlin Heidelberg, 1987.

[LaTeX] Helmut Kopka. *„LaTeX: Eine Einführung“*. 4. überarbeitete und erweiterte Auflage. Addison-Wesley (Deutschland) GmbH, Bonn, 1993.

[TeX] Donald E. Knuth. *„Computers and Typesetting Vol. A: The TeXbook“*. Addison–Wesley Co., Inc., Reading, MA, 1986.

Abbildungsverzeichnis

Index