The PalmOSnomicon[•]

Rainer Brockerhoff

<rainer@brockerhoff.net>
http://www.brockerhoff.net/

***Abstract**: A shadowy client hires Our Hero to write a Palm Application; the mad scribblings of a deranged programmer surface; Our Hero goes "by the docs" into the nether regions and narrowly avoids doom; the eldritch denizens of the Palm OS appear menacingly; St. Bjarne appears in a fever dream and mutters "pure virtual functions"; the ultimate doom is beaten back in the last moments by the power of Object-Oriented Programming and Our Hero survives to write another paper.*

A Shadowy Client

It was a dark and stormy night[1], and the acoustic effects in the gloomy meeting room were definitely unsettling. I squinted at the shadowy figure across the large table but no appreciable resolution enhancement ensued. I cleared my throat with some impatience. “Some fancy headgear you have there”, I commented. “First time I’ve ever seen a tentacled propeller beanie.”

“Beanie…?” he said in a puzzled tone. The thing on his head (or *was* it his head?) seemed to squirm unpleasantly. “But never mind. I hear you know something about Palm programming?” he asked.

“I’ve got loads of experience with embedded 68K processors; and I’ve got a copy of CodeWarrior for the Palm Pilot stashed away somewhere,” I said. “Not that I’ve had much time to look at it, though.” A roll of thunder was answered by a chittering noise behind me.

He went on, “We need this application done ASAP, and the fellow we originally had on the project met with — well, let’s call it a little trouble.” With that, he slid a battered-looking Palm across the table.

I’d never seen this model before. I poked at it with a finger. “Say, doesn’t this green goo on the touchscreen void the warranty? And what sort of trouble do you mean, anyway?”

The chittering behind me seemed to swell, and a flock of vaguely bat-like creatures flew from under the table to perch on the guy’s shoulders. He hissed at them in annoyance. “He delivered a

[•] Or "The Virtual Call of Cthulhu", or "Up Palm Crick Without a Stylus", or "12 Easy Steps to the Forms of Madness", or "Programming for the Palm OS With One Cerebral Hemisphere Tied Behind Your Back". Pick your favorite title!

beta version, but failed to get the shoggoths[2] out in time," he rumbled.

Figure 1 [3]

"Shoggoths… you mean bugs, I suppose? I want the original specs and the beta source code, then. You know my rates for a rush job?" His answer was drowned out by another roll of thunder, and some more goo splattered down from the ceiling. A rustling noise came from under the table. He drummed his fingers impatiently on the table, waiting for things to settle down. "Come on," he finally growled, "let's reset to the standard appearance. This is getting quite annoying."

"Oh well," I conceded, and fumbling the remote out of my pocket, pressed the 'default' button. The room brightened and the shadowy figure morphed into a rather large man in a business suit, the tentacles fading into somewhat unruly graying hair. The chittering was drowned by the hum of an air conditioner. "I suppose H.P. Lovecraft's estate wouldn't be too happy about this, anyway — and the readers probably won't like source listings in Pnakotic script. So what were you saying?"

He handed me a large folder with some relief. "Here are the specs, and the other guy's source code. He started out well enough but then couldn't handle it when we asked for revisions."

"I'll do my best," I promised.

The Dreaded Spaghetti Code

The next day, I looked at the existing code. When the original Palm Pilot came out I'd bought one in a package deal with a bundled copy of CodeWarrior[4]. Although I've never quite gotten around to doing something useful with either one, I had installed the updates and downloaded SDK version 3.5 from Palm's site[5].

(From here on, I'll assume that you have the latest[6] CodeWarrior and Palm OS SDK installed, as well as some experience with C/C++ and the Palm OS.)

Checking the other guy's code with the Palm OS documentation, as well as with the example apps in the SDK, I could see that something strange was going on. Where the example apps had several forms using standard user interface objects, his application had a single empty form and he inserted all UI objects dynamically whenever he needed to. The object's positions were hard-coded, and an example database was set up from an array of structures with a couple of pages of initialization text.

Tossing the listing away with some distaste, I decided to go "by the book" and redo the application completely, starting from the skeleton C application generated by the CodeWarrior stationery. Why C and not C++? Well, I instinctively shied away from imposing object and exception overhead upon a handheld application — also, all non-trivial examples I found were in C. A mistake, as we'll learn later on.

The general structure of the C skeleton is quite simple. We have a `main()` routine, an `AppHandleEvent()` routine, and an `AppEventLoop()` routine, as well as some others. I'll refrain from going into the details here, as they're quite well explained in the Palm OS documentation.

Everything looked quite reasonable to me, but as I'd never actually programmed for the Palm OS I decided to go the "gradual top-down" route. That is, I defined an initial form, set up the event handler stubs with a debugger break point, and verified that events were being received at the proper points. I then gradually wrote more complex actions into the stubs, asymptotically approaching whatever the program was supposed to do.

A Saintly Apparition

Without boring you with unnecessary details, I spent about two weeks designing forms and writing code. It all seemed quite straightforward, and while there were several tricky points handling the Palm's way of doing databases, or coping with the limited amount of dynamic memory available, everything seemed to be going well.

Nevertheless, as I slowly hacked away at the product specification, things began to slow down. Nearly every form seemed to be a list of something or other, but there were subtle differences between each list. I implemented those using table objects. Most of the lists had more lines than could fit on the screen, so I had to consider scroll bars and switch the table data around according to what was visible. The table API is very complex and many calls have to be made, declaring item handlers, marking rows and columns as visible, and so forth.

This was especially aggravating since my table had only one column, a variable number of rows, and the standard text item format did not look like I wanted it to. For instance, my main form initialization routine at this point looked as follows:

```
void MainFormInit
  (FormPtr frmP) {
  Int32 i;
  theTable = GetObjectPtr
    (MainNumberTable,frmP);
  rows = TblGetNumberOfRows(theTable);
  TblGetBounds(theTable,&bounds);
  for (i=0;i<rows;i++) {
    TblSetRowUsable(theTable,i,false);
    TblSetItemStyle
      (theTable,i,0,customTableItem);
    TblSetCustomDrawProcedure
      (theTable,0,&DrawNumberItem);
    TblSetRowStaticHeight
      (theTable,i,true);
    TblSetRowHeight(theTable,i,24);
    TblSetItemStyle
      (theTable,i,1,labelTableItem);
  }
  TblSetColumnUsable(theTable,0,true);
  TblHasScrollBar(theTable,true);
  CheckAndRedrawMainForm();
}
```

Of course, I had to write a custom table item draw handler (`DrawNumberItem()` (), and rather complicated scrolling and updating routines.

As things progressed — or, rather, failed to progress satisfactorily — I found myself battling the same bugs over and over again, in slightly different variations. Finally, after a particularly unproductive weekend, I was ready to throw in the towel. Working over the code while tossing and turning in bed —

something which endears me no end to Vitamin C manufacturers, but not to my wife — I sank into an uneasy sleep. If dreams only had Internet access capability…

When I woke in the morning, I had a vague memory of St. Bjarne Stroustrup appearing to me and muttering something Scandinavian; the only phrase I could remember was "pure virtual functions". Hmmm… perhaps I had been barking up the wrong tree with my insistence on C?

I rushed to the computer, hooked up a caffeine IV drip, and set the "Activate C++" and "C++ Exceptions" flag in my project preferences. Fortunately I tend to code C in a style which avoids most things not understood by a C++ compiler, so after a few minutes of editing everything compiled and ran OK. Sprinkling some throw/catch pairs around the initialization code didn't seem to increase object code size appreciably.

The Magic of Objects

Looking again at my several screen forms, I first decided to implement a general `List` class with subclasses handling each special case. To get maximum flexibility I would use a Gadget form object to represent the whole list on the form, instead of using tables. Since all lists were single-column lists, I would implement a pure virtual `DrawItem()` function which every subclass would implement to draw that specific subtype of list item.

With some abridgements, my first try at the `List` class declaration looked like this:

```
class List {
public:
                List(
                FormPtr frmP,
                UInt16 gadgetID,
                Int16 maxItems);
  virtual       ~List();
  virtual Int16 SetCurrent(
                Int16 theItem);
  Int16         GetCurrent() const;
  Boolean       Draw();
protected:
  static Boolean ListGadgetHandler(
                FormGadgetType *gP,
                UInt16 cmd,
                void *paramP);
  virtual void  DrawItem(
                Int16 theItem,
                RectangleType& rect,
                Boolean selected)=0;
  Int16         mMaxItems;
  Int16         mCurrentItem;
}
```

In the public part, there's a constructor which gets the form pointer, the ID of the gadget form object, and the maximum number of items in the list. There are accessor functions to set and get the currently selected list item, and a `Draw()` function which redraws the list whenever needed.

In the private part there's a static event handler for the gadget, which just calls the current list's `Draw()` function, and the pure virtual `DrawItem()` function.

Having written this, I proceeded to write the appropriate `List` subclasses for each form. I won't go into details; there was some backing, filling, tweaking, and hacking, mainly to handle list scrolling, but all in all everything seemed to be working out fine this time…

…until, at our weekly meeting, my client suddenly said: "How about a pop-up menu right here…?" I squinted to see what he was pointing at. "What, right on my secondary auxiliary infra-sub-detail list??" I yelped. "Absolutely no way, or rather, that would entail a substantial delay, not to speak of the budget overrun!"

He looked annoyed. “Shouldn’t that be just an easy afternoon’s work with those marvelous virtual object thingies you were so enthusiastic about? And it would be much easier to use, you know; you’ll get fewer support calls.”

“Give me a day to think about it, then,” I conceded reluctantly.

Objectively Doing Generic Objects

Back at my computer, I considered how to cope with this newest spec change. After some fiddling around with the standard Palm popup objects, I realized that a redesign of my basic object hierarchy would be more productive — and coincidentally would also simplify my interface to scroll bars.

In the new scheme of things, I wrote a generic `Object` class to encapsulate a generic Palm user interface object. My former `List` class would become a subclass of `Object`, and other subclasses would encapsulate scroll bars and popup menus. After a sleepless night, my base class looked like this:

```
enum objRes {
  objNotHandled,
  objHandled,
  objBreak
};

class Object {
public:
              Object
              (FormPtr frmP,
              UInt16 objectID,
              Boolean show=true);
              Object
              (RectangleType* rect);
  virtual     ~Object();
  virtual void SetBounds
              (RectangleType* bounds);
  virtual void GetBounds
              (RectangleType* bounds)
              const;
  virtual Boolean Draw();
  virtual void Show();
  virtual void Hide();
  Int16       GetFormObjectIndex()
              const;
  virtual objRes HandleEvent
              (EventPtr eventP,
              Object*& object);
  Object*     GetObject();
  static Boolean HandleObjectChain
              (EventPtr eventP);
  static Object* GetDoubleTap();
  static void SetLastDown
              (Object* object);
  Boolean     DoubleTap();
protected:
  RectangleType mRect;
  FormPtr     mForm;
  void*       mObject;
  Int16       mObjectIndex;
private:
  Boolean     PtInRect
              (Coord x,Coord y);
  Object*     GetNextObject()
              const;
  Object*     mNext;
  Object*     mPrevious;
  static UInt32 sLastMove;
  static Object* sObjectChain;
  static Object* sLastDown;
  static Object* sDoubleTap;
};
```

(And yes, I promise this is the actual latest working version!) Let’s look at each part of this in turn. The `objRes` enumeration is used to define the return type of the event handlers; more about this later on. There are two `Object` constructors: one for encapsulating a Palm user interface object, and one for simply defining a rectangular part of the display as a custom interface object. The first constructor is as follows:

```
Object::Object
  (FormPtr frmP,
  UInt16 objectID,
  Boolean show) {
  mForm = frmP;
  mObjectIndex = FrmGetObjectIndex
    (mForm,objectID);
  if (mObjectIndex<0) {
    throw(appErrorClass);
  }
  mObject = FrmGetObjectPtr
    (mForm,mObjectIndex);
  if (mObject==NULL) {
    throw(appErrorClass);
  }
  mNext = sObjectChain;
  mPrevious = NULL;
  if (sObjectChain) {
    sObjectChain->mPrevious = this;
  }
  sObjectChain = this;
  FrmGetObjectBounds
    (mForm,mObjectIndex,&mRect);
  if (show) {
    Show();
  }
}
```

This is very straightforward. The object's index and pointer are obtained and stored; the new `Object` is inserted into the chain of objects pointed to by `sObjectChain`; the object's bounds are obtained, and the object is drawn if necessary. Let's now look at the other constructor:

```
Object::Object
  (RectangleType* rect) {
  mForm = NULL;
  mRect = *rect;
  mObjectIndex = -1;
  mObject = NULL;
  mNext = sObjectChain;
  mPrevious = NULL;
  if (sObjectChain) {
    sObjectChain->mPrevious = this;
  }
  sObjectChain = this;
}
```

This is even simpler, handling only the object chain, and setting other fields to null values. Other details are left to the derived classes' constructor. The Object destructor is:

```
Object::~Object() {
  Hide();
  if (sLastDown==this) {
    sLastDown = NULL;
  }
  if (sDoubleTap==this) {
    sDoubleTap = NULL;
  }
  if (mPrevious) {
    mPrevious->mNext = mNext;
  } else {
    sObjectChain = mNext;
  }
  if (mNext) {
    mNext->mPrevious = mPrevious;
  }
}
```

The destructor hides the object from view and unchains it from the object chain. Next, let's look at a whole bunch of shorter routines:

```
Object* Object::GetNextObject() const {
  return mNext;
}

Boolean Object::Draw() {
  if (mObjectIndex>=0) {
    FrmShowObject(mForm,mObjectIndex);
    return true;
  }
  return false;
}

void Object::Show() {
  if (mObjectIndex>=0) {
    FrmShowObject(mForm,mObjectIndex);
  }
}

void Object::Hide() {
  if (mObjectIndex>=0) {
    FrmHideObject(mForm,mObjectIndex);
  }
}

Object* Object::GetDoubleTap() {
  return sDoubleTap;
}

void Object::SetLastDown
  (Object* object) {
  sLastDown = object;
}

Boolean Object::DoubleTap() {
  return this==GetDoubleTap();
}

void Object::SetBounds
  (RectangleType* bounds) {
  mRect = *bounds;
  if (mObjectIndex>=0) {
    FrmSetObjectBounds
      (mForm,mObjectIndex,&mRect);
  }
}

void Object::GetBounds
  (RectangleType* bounds) const {
  *bounds = mRect;
}

Int16 Object::GetFormObjectIndex()
  const {
  return mObjectIndex;
}

Object* Object::GetObject() {
  return this;
}

Boolean Object::PtInRect
  (Coord x,Coord y) {
  return RctPtInRectangle(x,y,&mRect);
}

objRes Object::HandleEvent
  (EventPtr eventP,Object*& object) {
  return objNotHandled;
}
```

These are practically self-explanatory (I hope). The `HandleEvent` routine will nearly always be overridden by a derived class, of course. Finally, we'll look at the routine that makes everything work together: the event handler. This is critical, so I'll interpolate the explanations into the code, for a change.

```
Boolean Object::HandleObjectChain
  (EventPtr eventP) {
  Object* object=sLastDown;
  Object* next=NULL;
  UInt32 now=TimGetTicks();
  eventsEnum t=eventP->eType;
  switch (t) {
  case penMoveEvent:
    if ((now-sLastMove)<25) {
      return false;
    }
    sLastMove = now;
    // fall into next case
```

`penMoveEvents` are restricted to 4 per second, which will slow down things when the user drags the pen around.

```
  case penUpEvent:
    if (sLastDown&&
        (sLastDown->HandleEvent
        (eventP,object)==objHandled)) {
      sDoubleTap = NULL;
      return true;
    }
    sDoubleTap = NULL;
    return false;
```

Both `penMoveEvents` and `penUpEvents` are handled only if the same object received a `penDownEvent` before — as you'll see, the `sLastDown` variable always points at the last object which actually handled that event.

```
  default:
    object = sObjectChain;
    while (object) {
      next = object->GetNextObject();
      if ((t!=penDownEvent)
        ||(object->PtInRect
          (eventP->screenX,
          eventP->screenY))) {
```

Any generic event is handled by iterating down the chain of `Objects`, which are in order of creation — latest first. `penDownEvents` are handled only if the pen actually was inside the `Objects` boundary rectangle.

```
    switch
      (object->HandleEvent
        (eventP,object)) {
        case objHandled:
          switch (t) {
          case penDownEvent:
            sDoubleTap =
              (eventP->tapCount>1)
                &&(sLastDown==object)
                  ?object:NULL;
              sLastDown = object;
              sLastMove = now;
              break;
            case penUpEvent:
              sLastDown = NULL;
              break;
            default:
              break;
            }
            return true;
          case objBreak:
            sDoubleTap = NULL;
            return false;
          }
        }
        object = next;
      }
      break;
  }
  sDoubleTap = NULL;
  return false;
}
```

If a `penDownEvent` event has been handled, `sLastDown` will point at the object until it accepts a `penUpEvent`. The `sDoubleTap` variable will point to the object which received the last double pen tap; this will usually be checked by calling the `DoubleTap()` routine.

How does all this fit into a normal application? Here's a typical form event handler using a derived `ListObject` class:

```
Boolean MainFormHandleEvent
  (EventPtr eventP) {
  static ListObject* list=NULL;
  EventType evt;
  Boolean handled = false;
  FormPtr frmP = FrmGetActiveForm();
  switch (eventP->eType) {
  case penDownEvent:
  case penMoveEvent:
  case penUpEvent:
  case sclRepeatEvent:
    handled = Object::HandleObjectChain
      (eventP);
    if (list->DoubleTap()) {
      // handle double pen tap;
    }
    break;
  case frmOpenEvent:
    list = new ListObject(frmP);
    FrmDrawForm(frmP);
    handled = true;
    break;
  case frmCloseEvent:
    delete list;
    break;
  }
  return handled;
}
```

The advantage of using a basic `Object` class is now apparent. Derived classes need only override `HandleEvent()` to handle pen taps, and `Draw()` should they wish to provide some custom graphical appearance.

It Scrolls! It Slices! It Dices! With Optional User-Installable Self-Powering Field-o-Matic ® © ™ Attachment!

My former unsightly scroll handling was now simplified by encapsulating the scroll bars inside a `ScrollObject`, like this:

```
class ScrollObject:public Object {
public:
          ScrollObject
          (FormPtr frmP,
          Object* master,
          UInt16 objectID,
          Boolean ver,
          Boolean show=true);
  virtual ~ScrollObject();
  void    AdjustBounds
          (Boolean show);
  void    SetScroll
          (Int16 value,
          Int16 minv,
          Int16 maxv,
          Int16 psize);
  ScrollBarType* GetFormObject()
          const;
protected:
  virtual objRes HandleEvent
          (EventPtr eventP,
          Object*& object);
  Object*     mMaster;
  Boolean     mVertical;
};

ScrollObject::ScrollObject
  (FormPtr frmP,
  Object* master,
  UInt16 objectID,
  Boolean ver,
  Boolean show)
  :Object(frmP,objectID,false) {
  if (master==NULL) {
    throw(appErrorClass);
  }
  mVertical = ver;
  mMaster = master;
  SetScroll(0,0,0,0);
  AdjustBounds(show);
}

ScrollObject::~ScrollObject() {
  RectangleType rect={{0,0},{0,0}};
  SetBounds(&rect);
}

ScrollBarType*
  ScrollObject::GetFormObject()
  const {
  return (ScrollBarType*)mObject;
}

void ScrollObject::SetScroll
  (Int16 value,
  Int16 minv,
  Int16 maxv,
  Int16 psize) {
  SclSetScrollBar(
    GetFormObject(),
    value,minv,maxv,psize);
}

void ScrollObject::AdjustBounds
  (Boolean show) {
  RectangleType rect;
  mMaster->GetBounds(&rect);
  if (mVertical) {
    rect.topLeft.x += rect.extent.x;
    rect.extent.x = 7;
  } else {
    rect.topLeft.y += rect.extent.y;
    rect.extent.y = 7;
  }
  SetBounds(&rect);
  if (show) {
    Show();
  }
}

objRes ScrollObject::HandleEvent
  (EventPtr eventP,
  Object*& object) {
  switch (eventP->eType) {
  case penDownEvent:
  case penMoveEvent:
  case penUpEvent:
    return SclHandleEvent
      (GetFormObject(),eventP)?
      objHandled:objNotHandled;
    break;
  case sclRepeatEvent:
    return mMaster->HandleEvent
      (eventP,object);
  }
  return objNotHandled;
}
```

Now, part of my scrollable `ListObject` constructor does the following:

```
ListObject::ListObject
  (FormPtr frmP,
  UInt16 gadgetID,
  UInt16 scrollID,
  Int16 maxItems,
  Boolean show)
  :Object(frmP,gadgetID,false) {
  . . .
  if (scrollID>0) {
    mScroll = new ScrollObject
    (frmP,this,scrollID,true,false);
    mScrollIndex =
    mScroll->GetFormObjectIndex();
  } else {
    mScrollIndex = -1;
    mScroll = NULL;
  }
}
```

This attaches the `ScrollObject` to the `ListObject`, and keeps it always at tits right, even if the `ListObject` is moved around. Other `ListObject` routines also take the `ScrollObject` into account:

```
ListObject::~ListObject() {
  delete mScroll;
};

void ListObject::Show() {
  Object::Show();
  if (mScroll) {
    mScroll->Show();
  }
}

void ListObject::Hide() {
  Object::Hide();
  if (mScroll) {
    mScroll->Hide();
  }
}

void List::SetBounds
  (RectangleType* bounds) {
  Object::SetBounds(bounds);
  if (mScroll) {
    mScroll->AdjustBounds(false);
  }
}
```

and, finally, `ListObject::HandleEvent()` calls `mScroll->SetScroll()` with the appropriate parameters whenever a `sclRepeatEvent` is received, to implement live scrolling. The `sclRepeatEvent` handler also should return objBreak to pass the event to the Palm OS' scroll bar redrawing routines.

As a final example, let me show you how an editable text field would be implemented:

```
class EditObject:public Object {
public:
              EditObject
              (FormPtr frmP,
              UInt16 textID,
              MemHandle text);
  virtual     ~EditObject();
  FieldType*  GetFormObject()
              const;
  void        UpdateScroll();
  virtual objRes HandleEvent
              (EventPtr eventP,
              Object*& object);
  virtual void Show();
  virtual void Hide();
protected:
  ScrollObject*  mScroll;
};

EditObject::EditObject
  (FormPtr frmP,
  UInt16 textID,
  MemHandle text)
  :Object(frmP,textID,false) {
  FldSetText(
    GetFormObject(),text,4,
    MemHandleSize(text));
  mScroll = new ScrollObject
    (frmP,this,textID,true,false);
  UpdateScroll();
  Show();
  FrmSetFocus(frmP,mObjectIndex);
}

EditObject::~EditObject() {
  Hide();
  FldSetText(GetFormObject(),NULL,0,0);
  delete mScroll;
}

void EditObject::Show() {
  Object::Show();
  if (mScroll) {
    mScroll->Show();
  }
}

void EditObject::Hide() {
  Object::Hide();
  if (mScroll) {
    mScroll->Hide();
  }
}

void EditObject::UpdateScroll() {
  UInt16 scrollPos,
    textHeight,fieldHeight,
    maxValue,blkLines;
  if (mScroll) {
    blkLines = FldGetNumberOfBlankLines
      (GetFormObject());
    if (blkLines>1) {
      FldScrollField
        (GetFormObject(),
        blkLines-1,winUp);
      blkLines = FldGetNumberOfBlankLines
        (GetFormObject());
    }
  FldGetScrollValues
  (GetFormObject(),
    &scrollPos,&textHeight,&fieldHeight);
    if (textHeight>fieldHeight) {
      maxValue =
        textHeight-fieldHeight+blkLines;
    } else if (scrollPos) {
      maxValue = scrollPos;
    } else {
      maxValue = 0;
    }
    mScroll->SetScroll
      (scrollPos,0,maxValue,
        fieldHeight-1);
  }
}

FieldType* EditObject::GetFormObject()
const {
  return (FieldType*)mObject;
}

objRes EditObject::HandleEvent
  (EventPtr eventP,Object*& object) {
  Int16 delta;
  objRes result;
  switch (eventP->eType) {
  case sclRepeatEvent:
    delta =
      eventP->data.sclRepeat.newValue
      -eventP->data.sclRepeat.value;
    if (delta) {
      if (delta>0) {
      FldScrollField
        (GetFormObject(),delta,winDown);
      } else if (delta<0) {
        FldScrollField
          (GetFormObject(),
          -delta,winUp);
      }
      UpdateScroll();
    }
    return objBreak;
  case fldChangedEvent:
    UpdateScroll();
    return objHandled;
  case penDownEvent:
  case penMoveEvent:
  case penUpEvent:
    FldHandleEvent
      (GetFormObject(),eventP);
    return objHandled;
  default:
    result = FldHandleEvent
      (GetFormObject(),eventP)
      ?objHandled:objNotHandled;
  }
  return result;
}
```

Nearly all of this is straight out of the text field examples; a detailed explanation should be unnecessary.

Popup Magic

And what of my original Palm OS application? All of the various scrolling lists were easily rewritten as specialized `ListObject` subclasses. Even the dreaded popup menus were implemented as just another `ListObject`, with an interesting twist — my `PopupMenuList` constructor ends with:

```
Object::SetLastDown(GetObject());
```

which is very convenient, in that it allowed me to call `new PopupMenuList()` from the main list's `penDownEvent` handler.

If you'll examine the previous listings with some care, you'll see that the new popup object is thereby enabled to handle subsequent `penMoveEvents`. This in turn allows the user to place the pen on the main list to invoke the popup menu, and drag to one of the menu items; the menu's event handler destroys the popup when it receives the `penUpEvent`. Implementing a popup with standard Palm OS user interface controls would require three pen taps.

I'll spare you the sordid details about how I, flushed by my success, proceeded to implement buttons and dialogs using `Objects`.

Let's just say that the story had a belated happy end: the application came out well, if somewhat behind schedule, and several further last-minute requests were handled very quickly — all thanks to the power of objects, derived classes, and that sort of thing.

Acknowledgements

Many thanks to Andrew S. Downs, long-suffering editor/paper chair for MacHack 2001, for his invaluable suggestions, and of course to the hundreds[7] of fans who begged me to publish another paper.

Further References

The `Objects` I've detailed could, and perhaps will, in the future, be used as the basis for a simple Palm application framework… PalmPlant, maybe? Just kidding, Metrowerks! I've searched the Internet for frameworks (after I'd done all the work on my own — drat!) and found only the following two:

Teenee[8] is a freeware application framework for C++ programmers that focuses on usability and safe memory management.

Bear River's PAF[9] is an Enterprise application framework, especially for larger applications for business and government. It provides support for user interface widgets, streams, TCP/IP and scanning.

Both are much more complete, of course, and focus only in passing on user interface handling. They're free under certain conditions and should probably be looked at by anybody wishing to do a more complex application in C++ for the Palm OS.

---

[1] See http://www.bulwer-lytton.com/ for similar starting lines.

[2] Check out the H.P. Lovecraft archive at http://www.hplovcraft.com/ for the original gooey stuff.

[3] All similarities to extant clients, octopi, palms. Palm trees or Palm™s without a satirical purpose are purely coincidental and unintended. The Statue of Cthulhu is ©copyright by Steven Roach, whom I was unable to contact in any way. Palm Computing, Palm OS, Graffiti, HotSync, and Palm Modem are registered trademarks, and Palm III, Palm IIIe, Palm IIIx, Palm V, Palm Vx, Palm VII, Palm, More connected., Simply Palm, the Palm Computing platform logo, Palm III logo, Palm IIIx logo, Palm V logo, and HotSync logo are trademarks of Palm, Inc. or its subsidiaries. All other product and brand names are or may be trademarks or registered trademarks of their respective owners. The "Cthulhu Runes" font is ©copyright by Flat Earth, Inc, and may be downloaded for free from http://flatearth.com/fonts/cthulhu.zip.

[4] Found, of course, at http://www.metrowerks.com/.

[5] The latest SDK and other indispensable developer stuff can be found at http://www.palmos.com/dev/tech/tools/. You'll also find the documentation: "Palm OS Companion" and "Palm OS Reference" in PDF format.

[6] Actually, both CodeWarrior and the Palm SDK were updated just after I wrote the first draft of this; the updates may not be a 100% compatible with my code.

[7] "Hundreds", of course, referring to the binary representation of the actual number.

[8] See http://www.classactionpl.com/Teenee/index.htm for details about Teenee.

[9] See http://www.bearriver.com/developer/palm/ for details about Bear River's PAF.