应用设计模式设计可移植的、可扩充、简单灵活的嵌入式GUI Toolkit

佟立峰
(摩托罗拉中国公司全球软件组　北京100022)

Apply Design Patterns to Achieve Portable, Extensible and Flexible Embedded GUI Toolkit

TONG Li-Feng
(GSG-China, Motorola, Beijing 100022)

Abstract The environments in which embedded software run are different in that they may run on different processors, have different input devices, and display graphics on a variety of video devices. In addition, it usually has limited computing and storing capability which require embedded GUI toolkit simple and flexible. According to the characteristics and the requirements of embedded environment, this story shows an embedded GUI toolkit and some design concerns, that is, applying design patters to design embedded GUI Toolkit to achieve high portability, extensibility and flexibility.

Keywords Embedded, GUI, Design pattern

1 前言

在桌面计算领域Windows操作系统和x86系列CPU几乎一统天下，基于Windows提供的标准GUI控件，开发者们设计出了各种各样的应用程序，这些程序只要求能在配置了x86 CPU并安装了Windows操作系统的机器上运行。但是在嵌入式领域情况则大相径庭：软件需要运行在不同的处理器上，输入设备和作为输出的显示设备也千差万别。同时，嵌入式环境的计算、存储能力大都非常有限，因此，高可移植性、可扩充性和简单灵活就成了设计嵌入式GUI Toolkit的三个最主要的设计目标。

目前，在嵌入式GUI领域应用比较广泛的主要有如下几种：

· MicoroWindows/NanoX：MicroWindows在设计过程中采用了分层的思想，基本分为三层[5]：最底层是面向图形输出和键盘、鼠标或触摸屏的驱动程序；中间层实现了一个图形引擎，提供底层硬件的抽象接口，并进行窗口管理；最高层分别提供兼容于X Window和Windows CE(Win32子集)的API。

MicroWindows提供了相对完善的图形功能，但其图形引擎存在许多问题[4]，如无任何硬件加速能力，不能充分发掘硬件的性能；而且其代码质量较差，在图形引擎中存在着许多低效算法。

· OpenGUI：OpenGUI也分为三层：最低层是由汇编编写的快速图形引擎；中间层提供了图形绘制API，包括线条、矩形、圆弧等，并且兼容于Borland的BGI API；第三层用C++编写，提供了完整的GUI对象集。OpenGUI目前可运行在MSDOS，QNX，LINUX操作系统之上[8]，只支持x86硬件平台，可移植性稍差[4]。

· Qt/Embedded：Qt/Embedded是著名的Qt库开发商TrollTech发布的面向嵌入式系统的Qt版本，因为Qt是KDE等项目使用的GUI支持库，所以有许多基于Qt的X Window程序可以非常方便地移植到Qt/Embedded版本上。不过[4]，Qt/Embedded同样缺乏对硬件加速支持，而且Qt/Embedded的结构过于复杂，很难进行底层的扩充、定制和移植，尤其是那个用来实现signal/slot[7]机制的moc文件。

简而言之，这几种比较流行的嵌入式GUI由于多种原因在可移植性、扩展性和复杂度方面都或多或少地存在着问题。PowerParts是Motorola(更确切的说是其子公司Metrowerks)开发的一套针对嵌入式环境的GUI应用框架，本人参与了部分模块的设计和实现。在PowerParts的设计过程中我们从体系结构入手，应用一系列的面向对象设计模式很好地实现了这三个设计目标。

2 设计模式

模式(Pattern)并没有一个很严格的定义。一般说来，模式意指一种从一个一再出现的问题背景中抽象出来的问题的固定的解决方案。Christopher Alexander认为："每个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动"[1]。

设计模式指的是在软件的建模和设计的过程中所运用到的模式。Erich Gamma等人在《设计模式》一书中介绍了23个模式，从对象的创建，对象和对象间的结构组合以及对象交互这三个方面将模式分为创建型模式，结构型模式和行为模式[1]。本文将重点介绍如何应用设计模式设计可移植的、灵活的嵌入式GUI应用程序框架体系结构。

3 PowerParts的体系结构

PowerParts体系结构如图1所示：PowerParts的核心(PowerParts Core)是一个图形引擎，周围通过三个抽象层——硬件抽象层(Hardware Abstraction Layer)、RTOS抽象层(RTOS Abstraction Layer)、外观层(Appearance Layer)的封装使得内核与操作系统和硬件无关。

PowerParts核心主要实现三个功能：画图和裁剪、视图层次组织、事件处理系统和消息系统；硬件抽象层封装了所有

与输入和图形显示设备相关的代码，提供一个标准接口供PowerParts核心调用；RTOS抽象层封装了所有与操作系统相关的调用，使得PowerParts核心与特定操作系统API相互独立，事实上，因为PowerParts框架中包括了完整的作图原语和事件处理机制，它甚至可以运行在没有操作系统的裸板上；外观层提供了一套标准的外观类——Kauai，用户可以对Kauai进行替换和扩充以实现自己所需的外观。

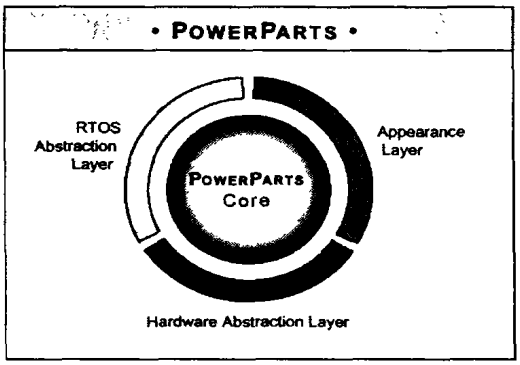

图1 PowerParts的体系结构

在上文提到的PowerParts的四部分的设计中，都应用了设计模式的思想以实现前文所提到的设计目标。

4 PowerParts中的设计模式

4.1 PowerParts核心层

在PowerParts核心层的设计主要应用了Composite[1]、Chain of Responsibility[1]和Observer模式[1]。

4.1.1 视图(view)层次组织中的Composite模式

Composite模式用来将对象组合成树形结构以表示“部分-整体”的层次结构。并且能够将一个组合对象简单化，将其作为单一对象处理。PowerParts核心层中的视图层次正是最为明显的一个Composite模式的应用。图2展示了PowerParts视图层次的类图。

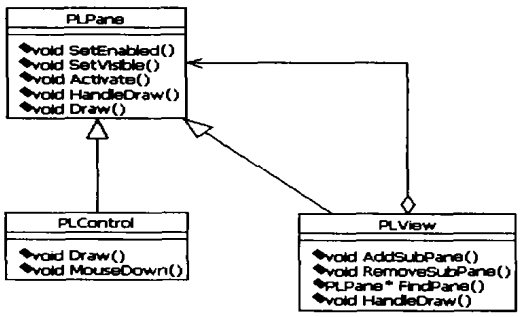

图2 Composit模式层次类图

在PowerParts中，PLPane[2,3]是最基础的可视化类，每个能画图的对象都是PLPane或其子类的实例。PLView[2,3]继承自PLPane，它可以包含其他PLPane或其子类的实例。PLView提供AddSubPane()，RemoveSubPane()，FindPane()等方法来实现对其所包含的SubPane进行管理。

图3为PLPane、PLView在一个典型界面中的应用。一个PLView对象通过包含其他PLView对象，从而形成一个复杂的视图层次。

视图层次负责维护坐标系、视图链中视图对象的相对关系和外观，这种设计使得实现诸如视图滚动这样的操作非常简单灵活，开发人员根本不必费心跟踪在哪里画什么，哪些区域可见，哪些区域不可见这样的问题。视图通常自己负责画出自己，在响应刷新事件时，根据视图包含关系由外层到内层依次执行画图操作。

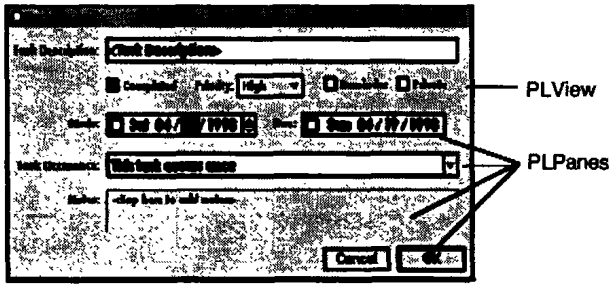

图3 视图层次示例

4.1.2 事件处理系统中的Chain of Responsibility模式

Chain of Responsibility模式将能够响应请求的对象连成一条链，事件沿着这条链传递直到有一个对象处理它。PowerParts的事件处理系统是采用Chain of Responsibility模式实现的，其类图如图4所示。PLEventHandler[2,3]是所有需要响应事件的类的基类。它提供了引入处理键盘、菜单等事件的能力，它的直接子类PLPane提供了处理鼠标事件的功能，每个PLEventHandler（或PLPane）的子类通过覆盖某些函数来响应事件。下面结合图5具体说明PowerParts中的事件处理机制：application负责管理target object，target object接收所有的事件。如果target object不能处理某个事件，事件被传递到事件层次中的下一个event handler：它的parent event handler。最终，如果没有child event handler处理此事件，最顶层的event handler（通常是application）收到并处理此事件。

Chain of Responsibility模式的应用简化了事件处理系统的设计：每个event handler可以有多个child event handler，但最多只能有一个parent event handler，因此事件由下至上传递只有一条通路，从而很容易地解决由谁来接收并处理事件这一事件系统设计的核心问题。

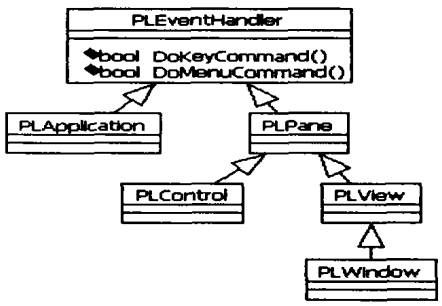

图4 Chain of Responsibility层次类图

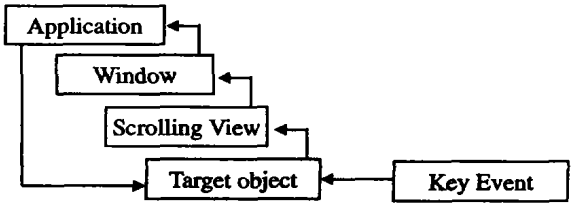

图5 事件处理层次

4.1.3 消息系统中的 Observer 模式 上一部分中介绍的事件处理机制对应用程序来说很重要，但是它有两个方面的局限性：

· 它要求进行通讯的两个对象必须在同一事件层次组织中

· 被处理的对象只能是事件

在许多情况下，程序员需要不同的对象可以自由地传递消息而不仅限于事件而且不必关心他们是否在同一事件层次组织中。正是在这种需求下，我们基于 Observer 模式设计了一个灵活的消息系统——Broadcast/Listen[2,3]系统，其类图如图6所示。

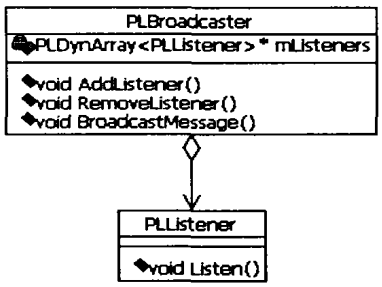

图6 Broadcaster/listener

Broadcaster 是一个可以在某种条件下发送消息的对象；Listener 是监听广播消息的对象。Broadcaster 调用 AddListener()方法注册 Listener 对象，当 Broadcaster 通过调用 BroadcastMessage()方法向所有注册的 Listener 广播消息时，Listener 接到消息并做出自己的处理，Listen()方法被执行。

例如，一个 checkbox 可以作为 Broadcaster，当它被选中时可以发出一个消息，Listener 接到消息后在 Listen()方法中可以做诸如更新窗体内某个控件状态的动作。

采用这种模式，Broadcaster 不必知道 Listener 将如何对消息进行响应等细节，同样，Listener 也不必知道 Broadcaster 的具体实现，它只需要理解所要响应的消息即可。这种松散的基于 Observer 模式的 Broadcast/Listen 结构使得消息系统具有极大的灵活性。

4.2 硬件抽象层中的 Facade 模式

硬件抽象层中的作用是使得 PowerParts 与输入和输出（视频）设备无关，它正是采用 Facade 模式[1]来实现此目的的，如图7所示。

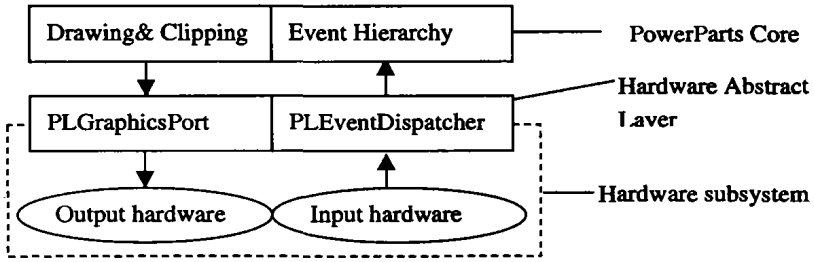

图7 硬件抽象层中的 Facade 模式

类 PLEventDispatcher[2,3]是硬件（或操作系统）将鼠标、键盘事件传递给 PowerParts 程序的接口。PLEventDispatcher 将鼠标、键盘事件插入到消息队列中，由消息队列决定哪个 event handler 应该接收此事件，即确定 target object 并将此事件传递给此 handler，此后由4.1.2部分介绍的事件处理层次将此事件传递给适当的 UI 元素。

PowerParts 采用 frame buffer 方式画图，PLGraphicsPort[2,3]作为 frame buffer 的抽象，通过定义宽度、高度、位深度和内存地址来描述不同的图形环境。所有的 Drawing& Clipping 操作都将对 PLGraphicsPort 所定义的 frame buffer 操作。程序员可以通过对 PLGraphicsPort 修改或者继承在一个设备上实现多种显示屏幕的支持或者使其更加适应硬件特征。

从上面可以看出，应用 facade 模式，把输入输出设备看作其封装的子系统，大大增加了系统的灵活性和移植性。

4.3 RTOS 抽象层中的 Facade 模式

Facade 模式在硬件抽象层的应用使得 PowerParts 与硬件做到了相互独立，同样 Facade 模式在 RTOS 抽象层中的应用使得 PowerParts 做到了与操作系统无关。嵌入式 GUI 与操作系统相关的部分主要涉及三个方面：线程（进程）、信号量和内存分配。PowerParts 通过提供 PLThread，PLMutex，PLSemaphore，PLMemoryAlloc，PLMemoryFree 等类形成了一个 Facade 层对 OS 所提供的服务进行了封装，其中，PLThread 实现了线程支持；PLMutex，PLSemaphore 实现了互斥和信号灯支持；PLMemoryAlloc，PLMemoryFree 封装了内存分配操作。当需要把 PowerParts 移植到其他操作系统时，只需要利用其操作系统提供的 API 实现这些 PowerParts 与操作系统的接口类即可。下面是 PLThread 的声明和在 PPSM-GT[6]操作系统中的实现（部分）。

```
class PLThread
{
  public:
        EXPORT void Start(int priority =1);
        //PURE VIRTUAL FUNCTION
        // This method contains the code to be executed in the
        newly created thread.
        // Subclass of PLThread needs to provide
        implementation for this method.
        EXPORT virtual int ThreadProc(void * data=NULL)
        =0;
        //……
  private:
        static int StaticThreadProc(void * data);
        //……
};
STATUS PLThread::Start(S8 priority) //task priority
{
  //……
  //Create a thread using PPSM-GT API
  status = KnlCreateTaskWith ( &taskID, ( P_VOID )
  StaticThreadProc,
  0, stackSize, (U32)this, priority, knlMode);
  //……
}
int PLThread::StaticThreadProc(void* data)
{
  PLThread* thread = (PLThread*)data;
  //execute thread code
  int result = thread->ThreadProc(thread->mThreadData);
  //……
```

}

4.4 UI 元素中的 Bridge 模式

核心以及硬件抽象层、RTOS 抽象层的封装使得 PowerParts 具备了很强的可移植性，这对嵌入式 GUI 应用开发人员很重要，但是对用户而言最重要的是 UI 是否美观适用。尽管 PowerParts 已经提供了一套完整、时髦的 GUI，但是由于嵌入式应用环境的多样性，不可避免地存在需要对其进行定制和扩充的需求，因此，对嵌入式 GUI 定制和扩充的难易程度就成了一项评价其设计优劣的重要指标。

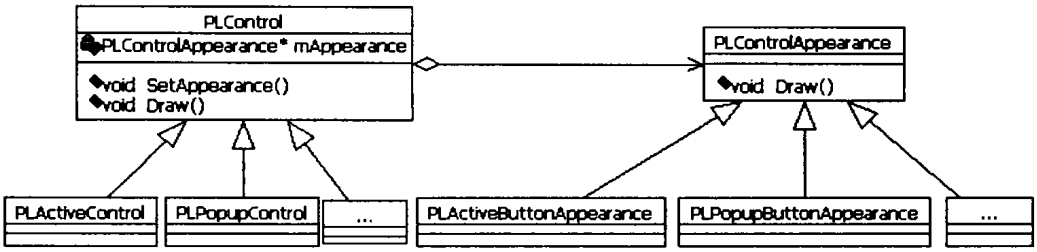

图8 控制层中的 Bridge 模式

如图8所示，PowerParts 在 UI 元素的设计过程中用 Bridge 模式[1]，将 UI 元素的行为和外观独立开来，形成各自的体系结构。图9和图10分别展示了 PowerParts UI 元素的行为和外观层次结构。

UI 元素的行为和外观可以独立地变化，使得创建、定制 UI 非常容易。举例来说，PLSliderControl 实现了 slider object 的行为而不是外观，每个 PLSliderControl 实例(slider)都指向同一个 PLSiderAppearance 对象，此对象负责画出所有的 slider，这种设计方式不但减少了内存消耗而且大大增加了可扩充性。如果需要实现一个不同的 slider，开发人员只需要继承 PLSiderAppearance 实现一个新的 slider 的外观而不必涉及 PLSliderControl。

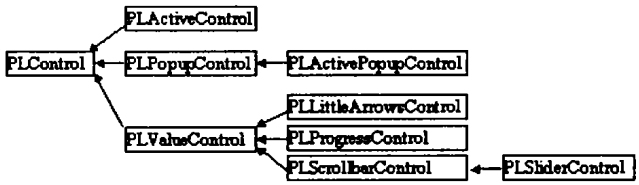

图9 控制层层次

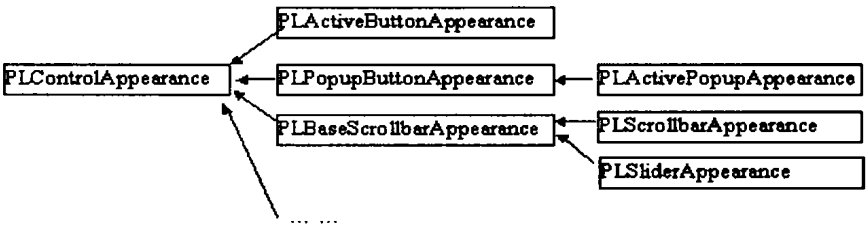

图10 控制外观层次

在此基础上，PowerParts 实现了对主题(Themes)的支持——一切换显示风格。

结论 使嵌入式 GUI Toolkit 具有高可移植性、可灵活定制扩充是设计嵌入式 GUI Toolkit 的主要目标。为了实现此目标，在 PowerParts 的设计过程中应用了大量的设计模式。本文结合 PowerParts 的层次结构，从宏观上介绍了 Composite、Chain of Responsibility、Observer、Facade 和 Bridge 模式在嵌入式 GUI 系统设计中的应用。当然，设计模式在 PowerParts 中的应用还不限于此，如动态数组中的 iterator 模式和字体类中的 strategy 模式。目前，PowerParts 已经成功地移植到 Windows，Linux，Macintosh，PPSM 等平台，支持 X86，68K 等多种 CPU。事实证明正是这些设计模式的成功应用使得 PowerParts 成为具有很高的可移植、可扩充性、简单灵活的优秀嵌入式 GUI Toolkit。

参 考 文 献

1 Gamma E, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

2 PowerParts ® Framework 3.0 User Guide. Metrowerks Corp. 2002

3 PowerParts ® Framework API Reference. Metrowerks Corp. 2002

4 魏永明. MiniGUI 和其它嵌入式 Linux 系统图形及图形用户界面支持系统

5 Haerr G. Microwindows Architecture. 2000/03/05

6 PPSM-GT 2.0 User Guide. Motorola Corp. 2002

7 唐新华. QT 的信号与槽机制介绍. 2001

8 OpenGUI--The Fast Graphics Library. 2000