论系统架构之松散与耦合

作者： fbib

这篇文章本是 2003 年华东狐上海会议的演讲稿。但是，整篇文章的思想在当时还没有成熟，有几个理论问题还没有找到更合理的解决方法，因此，上海会议的时候，我讲的自己觉得很不理想，参加上海会议的几个朋友帮我吹我连夜赶写演讲稿怎么怎么勤力，很惭愧，其实一方面是理论未成熟，难以下笔；另一方面是自己琐事缠身　　刚从法院打离婚官司回来，心情糟透了。L

在上海丢了人，自己很不爽。经过这一段时间的研究、体会，终于觉得能够拿出一点比较成熟的东西了，于是就写下来交给大家，算是“ 补考 ”吧！

最近一段时间，我接连翻译了两篇关于商业规则和商业对象的文章。要理解商业规则和商业对象，首先就必须理解为什么要将界面代码与数据代码相剥离，理解了这一点以后，才会认识到商业规则和商业对象的重要性，真正去研究和掌握它们。

过去，我们的程序中通常都存在大量数据代码和界面代码混合在一起的情况，由此直接导致了程序结构的复杂化，难以调试，也难以修改。我们急需一种理论的指导，让我们能够清晰的定义程序中各个部分的责任，使得程序的结构能够更加清晰、合理。

关于这方面的内容，一直没有很好的资料。我们听了很多关于商业规则、存储过程的东西，但是就是不明白为什么需要它们。通过一段时间以来的研究，我渐渐的觉得自己已经摸到了一点门径，也觉得有责任将它写出来，帮助所有想提高的 foxer 向一个更高的水平前进。

在我们以前的学习过程中，面对对象理论的学习可以说是第一个台阶；客户/服务器、三层应用可以说是第二个；而我现在相信：商业规则商业对象、应用程序框架的研究，绝对可以算是 foxer 学习中的第三次浪潮。

可维护性的需要

大多数人，在经过了 3 到 5 年时间的努力学习和实际编程、积累一定经验了以后，都能够独立开发一个或多个程序。程序的好坏姑且不论，至少，它能够、而且的确是满足了用户的需求了的。

于是，雄心勃勃的开始成立工作室、接单子、做项目，做了两三个项目以后，问题来了：用户不停的提出新的需求、或者要求做这样那样的改动。你不得不经常得在各个用户之间跑动，不得不花大量的力气去看懂自己的源代码，然后修改它。

由于编程的时候，只考虑怎么完成功能，而没有为将来的维护做好准备，结果，就

像玩 UML 的人说的那样，一开始的时候，只是准备搭一个狗窝，搭着搭着，一不小心搭成了一座摩天大厦。由于没有事先的统筹安排，这座大厦里面的结构是乱七八糟、混乱不堪的。同时维护这样的三四个系统就变成了一场绝对的噩梦。

想要用户不改动需求是不可能的，哪怕你在摸完需求之后全部写下来让用户签字画押也没用，实践证明的事实就是如此：用户往往只有看到你完成的程序真正运行起来之后才真正对自己的需求有一个大致的概念。而且实际情况往往比这更复杂，比如，一开始向你提供需求的可能是某个部门经理或者具体的软件操作人员，而真正决定最后的需求的则是客户公司的老板。即使你的软件通过了部门经理和操作人员的要求，老板仍然可能在最后关头提出新的需求。《凌波微步》一书中就谈到过这样的实例。

有时候，我们常常开玩笑说，一个软件通常就是要开发两次才真正算完，一次开发就能通过反而是个奇迹。

然而，修改程序程序的事情，往往不是加一点或者减一点代码这么简单的。由于编写代码时没有对系统架构预先的考虑，写代码的时候通常是不择手段的完成任务，而不是遵循一套预先的约定，这样就给系统维护带来了极大的隐患。

要解决这样的问题，可以从三个方面入手：一方面，应该先向客户提供一个快速原型，以便于真正摸清客户需求；一方面，应该学习一点 UML 来进行系统分析；最后，应该采用一套成熟、灵活的 Framework 来开发。在这套 Framework 中，应该为程序的维护提供一个有效的机制、以及足够的接口和准备。

自发的编程方法的缺点

我把大家在学习了编程以后，自发的形成的编程习惯称之为“ 自发的编程方法 ”。这种方法往往是以程序的需求为出发点，只求完成当前的任务，而没有进行系统的分析、没有经过系统的总结，没有上升到一种理论的程度。在这篇文章中，我要讲的就是关于这种编程方法的一个很大的缺点：耦合相当的严重。

例如，假定现在表单上有一个文本框 Text1，你在表单的事件或者方法中使用 This.Text1 这样的形式来引用这个文本框，而在另外的表单控件中则使用 this.parent.text1 、或者 This.Parent.Parent.Text1 这样的形式来引用它。而在这个文本框中引用表单上同样的控件也会采用同样的方式。

现在如果我要把这个 Text1 移动到一个 PageFrame 的 Page1 上，头疼的事情就出现了：你必须到处去找到那些引用这个控件的代码，把它们一一修改过来。然后，还要修改这个 Text1 对其它控件的引用。象 thisform.xxx 这样的绝对引用还好说，象 this.parent.xxx.xxx 这样的相对引用就麻烦了，因为即使是使用字符串搜索工具也很难找到这样的引用，稍不小心就可能遗漏、出错。

更糟糕的是类之间的相互引用，设计错误的某个类的某个方法中，可能会要求必须有另一个类、或者一个数据表存在，甚至要求数据表中必须有某几个字段存在、或者数据表必须具备某种结构。这样一来，类的重要特征　　可重用性　　就大大的被降低了。如果一个类只能适合于严格要求的某一种特殊条件下，那么这个类还有什么把它做成类的意义？

对象与对象之间、对象与数据之间，由于直接或间接的引用而造成的对彼此之间的依赖性称之为耦合。这个对象的功能可能必须依赖另一个对象的属性或值才能运行，这样，在这两个对象之间就存在着一种依赖性，它们之间就可以称为紧密耦合。如果两个对象彼此之间几乎没有什么依赖性，即使不知道任何对方的情况也可以自己独立完成功能，那么，我们说这两个对象之间就是松散耦合。

耦合的问题，看上去很小。但是，在你没有意识到这种情况的危害的前提下，程序中就有可能出现大量的耦合。你可以想象一下自己做了一个主页，然而有一天，你的主页要换一个服务器了，而且你准备更改一下主页的目录结构　　忽然，你就碰到了一个大问题：原来你的大量的链接都随着目录结构的更改而变得无效了…于是，你只得沮丧的去修正大量的无效链接，手工改的话，那绝对是一件繁琐而悲惨的活。

很显然，我们需要一种高效的方式，一方面，要尽量减少会随着“ 目录结构 ”的改动而变得失效的“ 链接 ”；另一方面，需要一种让我们能够一次就能成批的搞定大堆的无效“ 链接 ”的模式。

从维护程序的角度来说，就是我们需要尽量减少、甚至消除在修正无效耦合上所浪费的时间，使得自己能够从这些繁琐而低级的重复劳动中摆脱出来，将精力集中在业务规则的改动上，迅速的完成程序的改动。因此，对象之间的紧密耦合当然是越少越好，最好能够完全取消紧密耦合，当然，这是一种最理想的状态，你也许需要根据实际情况做出一些妥协。无论如何，减少耦合都是必须的。

解决耦合问题的理论基础

在上海会议的演讲中，我提出了一种以桥梁模型为理论基础的串连式应用程序框架结构。当时做演讲的时候，已经觉得这种设想还不够成熟，经过了一段时间的学习和研究之后，我终于认识到，解决耦合问题的最终依据早就存在于编程的基本理论之中：任务的合理分解、提高代码的集中管理程度、提高代码的可重用性。不过，桥梁模型是一个有助于理解概念的好例子，让我们先从它开始吧！

桥梁模型

我曾经翻译过 Foxtalk 杂志的一篇关于桥梁模型的文章，在继续下面的内容以前，大

家可以先找那篇文章熟悉一下桥梁模型的概念。

桥梁模型的目的，是“ 将一个抽象与它的实现进行耦合，这样两个部分都能变得动态的独立 ”。什么意思呢？这么说吧，我们以汽车打个比方，开车的人不需要知道发动机是怎样工作的，也不需要知道装在发动机罩下面的是哪种类型的发动机。他只需要知道使用汽车方向盘、油门、刹车杆等等各种功能的“ 接口 ”就行了。真正的变速、转向、刹车功能是不是做在方向盘、油门、刹车杆里面的？很显然，不是，它们是通过访问汽车发动机、变速器这样的系统来实现自己的功能的。

你可以给一个发动机配置不同的方向盘、油门、刹车杆，反过来也是一样，一个方向盘也可以用于各种不同的发动机。发动机不需要知道有关方向盘、油门之类东西的资料，方向盘、油门也不需要知道发动机的情况。这种特性，就是桥梁模型所称的“ 两个部分都能变得动态的独立 ”。

在发动机上，已经为连接方向盘、油门、刹车杆预留了接口，反过来在方向盘、油门、刹车杆上也是如此，因此，只要在装配汽车的时候将它们连接起来就可以了。它们两者之间的互动关系是通过它们的父容器　　汽车来建立的。

任务的合理分解

反过来想一想，如果把一个方向盘做成一个了解世界上所有的发动机的情况，并能根据这些发动机的不同而作出不同的反应的东西　　这将是怎样一个怪胎方向盘？我猜这个方向盘最后会做得比整辆汽车都大。被大多数人弃置不用的表单向导所用的 txtbtn 按钮组类就是这么一个“ 怪胎方向盘 ”。

过去，在没有认识到这个道理之前，这样的“ 怪胎方向盘 ”我也做过不少，比如什么能适应任何表结构的 Treeview 类、能适合任何情况的复合查询类、能满足任何要求的命令按钮组类、到处都能用的工具栏类等等等等…结果，往往发现最终这是一个 Impossible mission，只得放弃。不好意思的说，在我的硬盘上还有不少这样永远也无法完工的垃圾类。

问题的关键在于，我们以为类应该是能完成某个领域的所有任务的、我们以为程序应该做成真的就像搭积木那样只要简单的把类堆起来就可以。这就犯了一个根本性的错误。

类的建立，首先应该立足于合理的任务分解。每个类，只应该完成只跟自己有关、自己能独立完成的任务，而绝不应该插手不该自己插手的事情。试图做一个包办一切的类，往往正是耦合出现之源。

下面请大家看一个界面图。

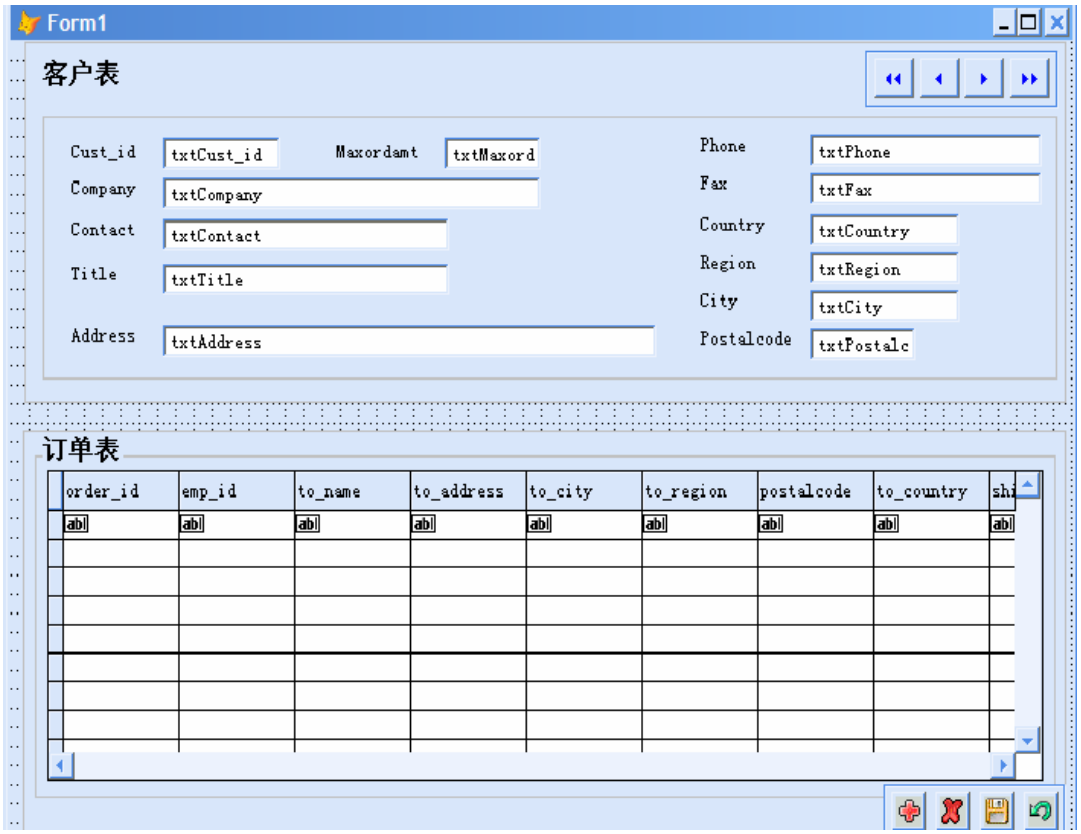


图 1、基于桥梁模型的客户　订单界面

这是一个典型的客户　　订单输入一对多表单。从这个界面上可以看到，表单被容器分成了两块：客户表部分和订单表部分。上面的部分是客户表容器，下面的部分是订单表容器。

客户表容器由两个子容器组成：输入控件容器，所有的数据绑定型控件都放在它里面，和一个用于移动记录指针的命令按钮组容器（通常我们把它叫做 VCR 类，以下都将命令按钮组容器类简称为 VCR 类）。

订单表容器也由两个子容器组成，一个是 Grid，另一个是常见的添加、删除、保存取消命令按钮组。

现在，我们试以这个表单中的 VCR 类为例，分解一下任务：

VCR 类：

- 在单击了某个子按钮时，告诉外部对象（表单，或者父容器），用户单击了某个按钮，要求移动记录指针；
- 根据相应的表（在这里是客户表）中记录指针的位置使能或禁止自己的各个子按钮。

先讲这个类的任务是有用意的。仔细思考一下这个类的任务，会发现一点微妙的东

西。

VCR 类其实并不应该直接去用 Skip、Go 命令去移动客户表中的记录指针。因为移动记录指针的任务其实并不象想象中的那么简单：

- 在真正移动记录指针之前，必须要先检测当前表单（包括客户表和订单表）上是否有未提交的改动。否则，如果某个表中打开了行缓冲，那么就可能弹出一个“ 表单上含有未保存的缓冲 ”错误。
- 如果有改动，则需要提示用户，并根据用户的选择来进行保存或放弃改动操作；
- 在保存的过程中还可能会因为数据缓冲冲突而造成保存失败。此时，则需要提示用户，再根据用户的选择去执行覆盖冲突的数据、返回原状态等待用户处理、或放弃所有用户的改动之类的操作；
- 最后，还要让子表（订单表）也根据客户表记录指针的移动而刷新自己的数据…

你会怎么办？把所有的这些检测数据更动、保存/放弃改动、处理缓冲冲突、刷新子表数据的代码都集成在这个 VCR 类中吗？不会吧？除非阁下最近手头非常紧，到肯德基麦当劳打工专做巨无霸汉堡包…J

很明显，上面的这四项任务，都不是按钮组自己的活！都不该由按钮组自己来做！

在按钮组之外，表单上应该有一个“ 数据对象 ”的存在，这个数据对象可以是表单、可以是客户表容器、也可以是客户表的 CursorAdapter 或者某个 Custom 类　　总之，不管是谁，总得有一个对象把那些上面四个该死的检测、操作数据的任务，甚至包括移动记录指针的任务拿走！

现在，假定我们已经有了一个数据对象，它接手了上面四个任务加上真正移动记录指针的活。好了，接下来再定义一下这个 VCR 类的输入和输出，可以帮助我们更清楚的决定这个类的任务。

输出：

- 向数据对象报告，要求移动记录指针。

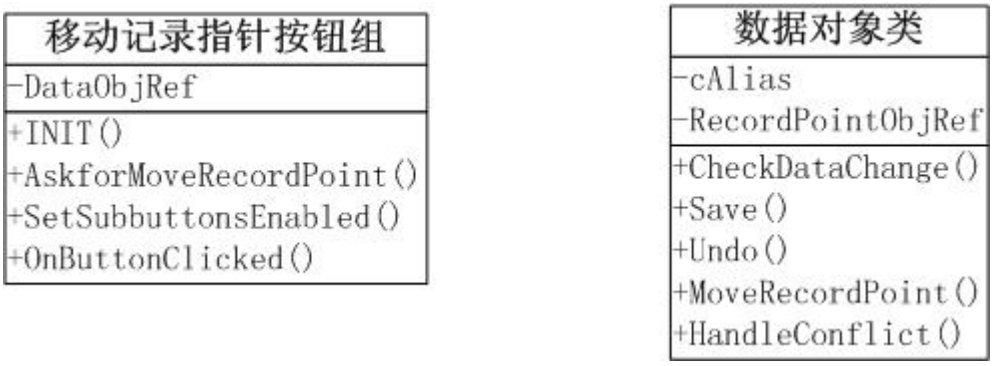
输入：

- “ 数据对象 ”移动记录指针成功，检测到记录指针现在处于表中的什么位置，并把它告诉按钮组；
- “ 数据对象 ”移动记录指针失败，告诉按钮组一切保持原样，什么也别动；这一项可以省略，因为除非数据对象向按钮组发送移动了记录指针的通知，否则按钮组保持原样不动。

这样一来，按钮组的任务就简化了、也明确了，它只有两个简单的任务：

- 向数据对象报告，要求移动记录指针。
- 根据数据对象发送过来的移动了记录指针的通知，设置自己各个子按钮的使能和禁止。

现在，我们可以清楚的定义类应有的属性和方法了，见下图：



这是一个 UML 类图，左边的是 VCR 类，可以看到它分成了三部分，最上面的一栏是类名，中间一栏是属性，我在这里给类添加了一个 DataObjRef 属性，用于记录下数据对象的引用名，比如一个字符串¡±thisform.oDataObj¡± 第三栏是方法，可以看到这里面有四个方法：

- INIT 事件大家都知道什么意思，具体的任务我下一步再说，
- AskForMoveRecordPoint()方法执行第一个任务，就是向数据对象请求移动记录指针；
- SetSubbuttionsEnabled()方法完成第二个任务，根据接收到的一个状态值参数（到头、上一条、下一条、到底）设置自己的各个子按钮的使能和禁止；
- OnButtonClicked()是一个事件，当用户单击某个按钮时发生；

注意这个类，可以观察到一点重要的东西　　在这个类里，没有任何跟数据有关的代码。这个类只处理与自己的各个子按钮有关的状态设置，也就是说，这个类是与数据无关的、与任何数据表都没有耦合的地方，它实现了将界面代码与数据代码相剥离的目标。

真正跟数据有关的代码都在右边的“数据对象类”里面。这里有五个方法，CheckDataChanged()检查当前是否有未保存的改动；Save()保存改动；Undo()放弃改动；HandleConflict()处理缓冲冲突；MoveRecordPoint()里则是真正移动记录指针的代码。

移动记录指针类负责管理自己的各个子按钮，数据对象负责实际的数据表操作。移动记录

代码集中管理

在上面的例子中，还有一点没有讲到的东西，有的朋友可能已经想到了：VCR 类里面的各个按钮呢？它们的代码是怎么样的呢？

答案是：这些按钮没有代码。

除了给这四个命令按钮各自设置了一个图片以外，我的确还做了一些事情，我给它们的每个按钮设置了一下 TAG 属性，分别为字符串 TOP、PREVIOUS、NEXT 和 END。

接着，我在 INIT()事件中利用了 VFP8 的 Bindevent()：

```
VCR 类.INIT()
  ************************************************************************
  local oSubButton as object
  for each oSubButton in this.buttons
      = bindevent(oSubButton, "Click", this, "OnButtonClicked")
  next
  ************************************************************************
```

Bindevent()的第四个参数被称为“代理”，就是说，四个命令按钮上都没有 Click 事件代码，那么，一旦其中一个发生了 Click 事件的时候，要执行的任务就由这个按钮组的 OnButtonClicked()事件来代理。

这就是代码的集中管理。

不管是面对对象还是面向过程，尽量提高代码的可重用性，以最大限度的减少重复劳动，是自有编程以来不变的原则。面向过程的办法是将可能被重用的代码建立成函数，面对对象的办法是建立类，而 BindEvents()又向我们提供了一种新的方法：将相似的代码集中起来管理。

由于有了 BindEvents()，我们就可以将代码集中在一个地方而不是在每个按钮按钮中各写一遍，一旦需要对代码做出修改，我们也就只需在 OnButtonClicked()中修改一次，而不需要在四个按钮中重复的修改四次，其优越性是很明显的。

下面是集中在 OnButtonClicked()方法中管理的代码：

```
移动记录指针按钮组.OnButtonClicked()
************************************************************************
LocaloSourceObj as object,oDataRef as object, cTag as string

** 取得发生 Click事件的按钮对象引用
=Aevent(aSourceObj, 0)
oSourceObj = aSourceObj[1]
```

```
** 返回该按钮对象的 TAG属性值
cTag = oSourceObj.TAG

** 让数据对象去处理
** 注意下面的做法是有问题的
oDataRef= EVAL(this.oDataRef)
IF VARTYPE(oDataRef) # "O" OR ISNULL(oDataRef) ;
      OR !PEMS(oDataRef, "MoveRecordPoint", 5)
      Return
END IF
oDataRef.MoveRecordPoint()
```

为什么说上面代码中的最后一部分是有问题的呢？因为它假定了有某个 oDataRef 对象存在，并且，该对象必须有一个名为 MoveRecordPoint()的方法。于是，除非真有一个带有这个 MoveRecordPoint（）方法的对象存在，否则这个按钮组类就不能工作，这就造成了耦合。

那么应该怎么做呢？让我们再想一下桥梁模型，桥梁的两端都可以动态的独立，只有在真正架桥的时候，才为两端指定相互之间的连接。也就是说：耦合不能存在于类里，否则类就丧失了通用性，但是耦合可以存在于类的实例里面。

VCR 类不应该知道任何有关数据对象、以及数据对象上哪个方法才是用来移动记录指针的。指定数据对象和数据对象上的某个方法应该在类被放到一个具体的表单上才去指定。类，是一个封装的层次，它应该具有通用性。而放到一个表单上以后，所存在的，是一个类的实例，这时再具体指定耦合的对象，其问题就小的多。

因此，在 VCR 类中的代码应该是这样的：

```
VCR 类.OnButtonClicked()
*********************************************************************************
Local oSourceObj as object,oDataRef as object, cTag as string

** 取得发生 Click事件的按钮对象引用
=Aevent(aSourceObj, 0)
oSourceObj = aSourceObj[1]

** 返回该按钮对象的 TAG属性值
cTag = oSourceObj.TAG

** 以上与前面的方法相同

** 类中的处理方法
Return cTag
```

只是简单的把 cTag 返回了，这能解决问题吗？嗯？

再看看类被放到表单上以后，类的实例的代码：

```
form.VCR 类 1.OnButtonClicked()
*********************************************************************************
 lcalcVal as string
 cVal = dodefault()
 thisform.oDataRefMoveRecordPoint(cTag)
*********************************************************************************
```

好了，这个 VCR 类现在就完工了。其实它相当简单，是吗？

复杂一些的例子

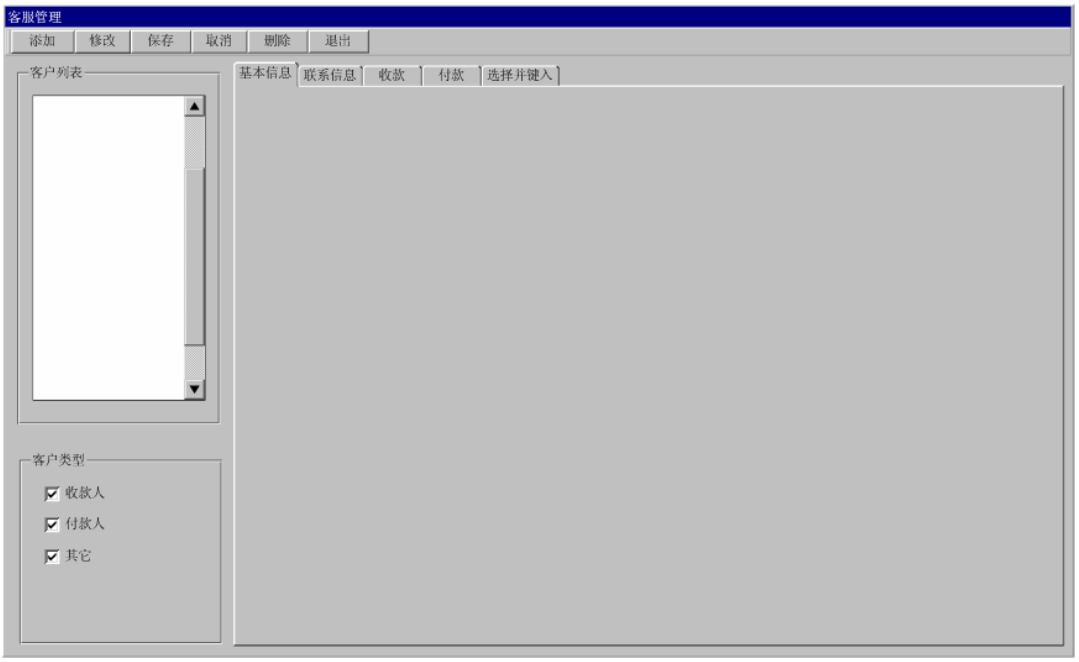
在简单的背后，是一些绝对不容忽视的编程原则。

现在，这个类的结构、代码都是非常简单的，而且，其中绝对没有耦合，因此，它是真正能够被放到任何表单上使用的。这是因为我们对它做了认真的分析之后，合理的为它分配了最适合它来做的任务，而将不属于它的任务全部都剔除了。

不同性质的任务，应该最合理的被分配给最适合执行这个任务的对象。类的设计原则，是尽可能的简化，而不是做成大而全的怪物。我希望大家都象我一样，将任务的合理分析和分解放在一个非常重要的位置。通过这个过程，可以最大限度的减少类中的耦合、从而提高代码的可重用性。理解了这个道理，也就为理解为什么要将程序结构分解成界面层和数据层打开了大门。

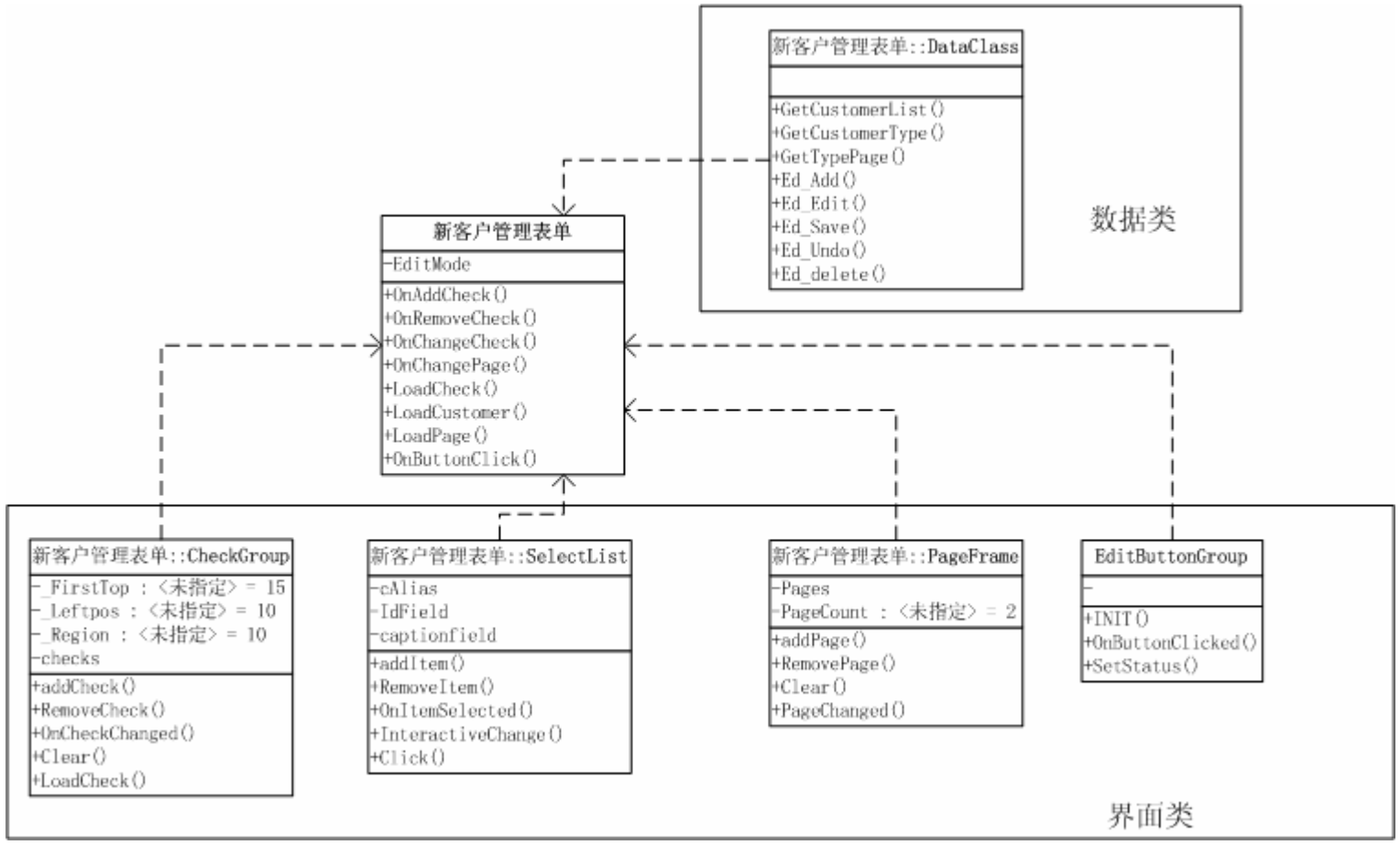
前面的例子你可能认为太简单了，不足以代表普遍性。那么，下面是我自己的一个客户管理程序中的主表单，我解说一下其大致的功能后，请自行思考一下怎样进行任务的分解：

客服管理主界面



最上面，是一个常见的编辑数据的命令按钮组；左边上面是一个客户列表框，用于显示数据库中有哪些客户；左下是一个客户类型的复选框组，右边是一个可以动态添加页面的页框，前面的基本信息和联系信息两页是固定的，而收款、付款页则是根据客户类型复选框中的相应的复选框是否选中而添加、删除的。

自己动手画一下，看看跟我下面的任务分解图有什么不同？


新客户管理表单::DataClass
+GetCustomerList()
+GetCustomerType()
+GetTypePage()
+Ed_Add()
+Ed_Edit()
+Ed_Save()
+Ed_Undo()
+Ed_delete()
数据类
新客户管理表单
-EditMode
+OnAddCheck()
+OnRemoveCheck()
+OnChangeCheck()
+OnChangePage()
+LoadCheck()
+LoadCustomer()
+LoadPage()
+OnButtonClick()
新客户管理表单::CheckGroup
-_FirstTop : <未指定> = 15
-_Leftpos : <未指定> = 10
-_Region : <未指定> = 10
-checks
+addCheck()
+RemoveCheck()
+OnCheckChanged()
+Clear()
+LoadCheck()
新客户管理表单::SelectList
-cAlias
-IdField
-captionfield
+addItem()
+RemoveItem()
+OnItemSelected()
+InteractiveChange()
+Click()
新客户管理表单::PageFrame
-Pages
-PageCount : <未指定> = 2
+addPage()
+RemovePage()
+Clear()
+PageChanged()
EditButtonGroup
-
+INIT()
+OnButtonClicked()
+SetStatus()
界面类

下面的四个界面类都只关心自己的事情：

- 复选框组类 **CheckGroup**

  只管加载、添加、删除、清楚全部复选框、以及当用户单击一个复选框时触发一个事件，虽然有一个后台表记录着有哪些可用的复选框选项（付款、收款、其它等等），但是在这个复选框组类上根本与该表没有任何关系。

- 页框类

  只管添加、删除一个页、或者清除所有的自定义页（VFP8 的新功能让我们可以建立自己的 Page 类，如果在 VFP8 以前，则只有将想放在一个自定义页上的东西做成一个容器，在添加了一个页以后，再将该容器放到新增的页上了。）、以及在用户单击选中了一页时触发一个事件 PageChanged。

- 选择客户的的 **SelectList** 类

  不知道将会作为它的数据源的是哪个表。当该类被放到表单上以后，需要设置一下它的三个属性：cAlias 是数据源表名，Idfield 属性记录在数据源表中作为编号字段或者主关键字字段的字段名，CaptionField 属性记录在数据源表中作为名称的字段（在这个表单上就是客户名称字段）。

  这个类执行的任务只有在用户单击一个列表时触发一个 Click 事件（不直接使用 InteractiveChanged 而用 Click 的原因是：我们需要在 InteractiveChanged 之前就检查当前表单上是否有未保存的更动以及一系列类似于 VCR 类中的情况，因此，我切断了 Click 就必然会引发 InteractiveChangd 的事件序列，而是让表单的 LoadCustomer 方法去决定要不要触发这个列表框的 InteractiveChanged 事件）、添加一条数据项（在这个表单上就是添加一个客户）、删除一条数据项（删除一个客户，通常用于用户添加了一个客户后未保存就选择了取消）。

- 编辑按钮类

  这个类的情况与 VCR 类类似，大家可以仔细研究一下。

表单　　中介者模型和发送信号的事件机制

中介模型的目的，是为了给一系列类似的对象提供一个中转站。通过让这些对象们之间相互的所有通讯都通过该部件或者子系统中的某一个部分来中转，这些对象之间的相互依赖性和关联就可以大大的减少。根据《设计模型》一书的解释，中介模型的意图，是要“ 定义一个概括了　一系列对象之间的交互　的对象 ”。

在表单上引入中介者模型的意义，在于将所有的耦合都集中在表单上，将原来在表

单的各个子对象之间的直接相互引用，变为通过表单的间接引用。这样做的好处，同样是减少了耦合，程序的修改和维护也更为容易。

例如，按我们以前的做法，如果正在编辑数据的过程中单击了命令按钮组中的“保存”按钮，那么，就必须在这个按钮的 Click 方法中这样写：

```
form.EditButtonGroup.cmdSave.Click():
*****************************************************************
    IF !thisform.DataClass.Ed_Save()
        && 可能会因为缓冲冲突而保存失败，
        && 而用户又选择返回
        Return
    Else
        && 保存成功，设置各个命令按钮的 Enabled状态
        This.parent.SetStatus()
    END IF
*****************************************************************
```

首先，这种做法就在这个按钮和数据类 DataClass 之间建立了耦合，如果该命令按钮组中的其它几个按钮都这样做的话，那么【添加】按钮还可能会与客户列表框 SelectList（添加一个客户时，需要向客户列表中添加一条新客户项）和客户类型复选框组 CheckGroup（新客户的类型尚未指定，需要将复选框组中的所有复选框还原成未选中状态）耦合，而删除、取消按钮同样也可能与前述 SelectList、CheckGroup 两个类相耦合。

一旦我想要改动程序　　比如将客户列表类 SelectList 换成了一个 Treeview 类，头疼的事情就来了：光是在这个命令按钮组类里面我就必须去修改三个按钮（添加、保存、取消按钮的代码都需要做修改了）的代码各一次，以将原来引用的 SelectList 换成 Treeview。而需要做这样修改的，还不仅仅是命令按钮组而已，别的对象中可能同样存在着这样的问题。

我们的解决办法，是象 VCR 类那样，首先，给每个命令按钮用一个关键字设置一下 TAG 属性，然后，在 INIT 中用 Bindevents()将所有按钮中的代码集中在一起，以该按钮组的 OnClicked()事件来代理，最后在将该类放到表单上时指定一下 OnClicked()事件的代码:

```
类代码 EditGroup.Init()
*********************************************************************
With this
    .cmdAdd.Tag = "ED_ADD"
    .cmdEdit.Tag = "ED_EDIT"
    .cmdSave.Tag = "ED_SAVE"
    .cmdUndo.Tag = "ED_UNDO"
    .cmdDel.Tag = "ED_DELETE"
    .cmdExit.Tag = "EXIT"
```

```
Endwith
for each oSubButton in this.buttons
    && 光这里就至少减少了多次耦合
    = bindevents(oSubButton, "Click", this, "OnClicked")
next
*****************************************************************************
```

类代码 EditGroup.OnClicked():

```
*****************************************************************************
local oSourceBtn as object, cTag as string
=AEVENT(aSource, 0)
oSourceBtn = aSource[1]
cTag = oSourceBtn.Tag

Do case
    Case INLIST(cTag, "ED_ADD", "ED_SAVE", "ED_UNDO", "ED_DEL")
        Return cTag
    Case cTag == "ED_EDIT"
        && 这里假设只是简单的设置一下命令按钮的 Enabled状态
        && 如果你需要做的更复杂，例如禁用表单上的 SelectList
        && 则可以按前面三个按钮同样的方法处理
        This.SetStatus()
        Return ""
    Case cTag == "EXIT"
        Thisform.Release
        Return ""
ENDCASE
*****************************************************************************
```

该类被放到表单上以后 Form.EditGroup1.OnClicked():

```
*****************************************************************************
local cTagRet as string
cTagRet = dodefault()
IF !empty(cTagRet)
    Thisform.HandleSignal(cTagRet)
END IF
*****************************************************************************
```

现在，这个类、以及这个类放到表单上的实例里面，只有唯一的一个耦合对象：表单。表单的 Release 方法是肯定存在的，因此写在了类里面也没关系。而 HandleSignal 则根据不同的表单，其名称也可能会不同，也许在下一个表单里就叫做 HandleButtonClick 了，因此，我将这个方法名称的引用放在了类的实例里面而不是类里面（注意这一点的不同 !），在不同的表单上可以根据需要修改。

其次，必须要注意的是，这个类只是在用户单击一个命令按钮的时候触发了一个事

件 OnClicked()，通过该事件，向表单发送了一个信号 cTagRet。至于触发了事件以后，操作（添加、保存、取消、删除）是否成功、以及根据是否成功来设置本命令按钮组成员的 Enabeld 状态，则根本不去管它。

请再回头看一遍刚才我们作为比较的错误代码，它是直接去调用表单上数据类的 Ed_Save()方法，然后，根据该方法的返回值来决定是否要 setStatus()。而在我们上面的办法中，这个类根本就不去管怎么调用 Ed_Save 方法和该方法返回的值是什么的事情。

从本质上来说，错误代码体现的，还是一种面向过程的思路：一连串的方法调用一直执行到底。而新办法体现的，则是真正面对对象的思路，用户的操作导致触发了一个事件，至于怎么处理这个事件就不关这个事件自己的事情了。面向过程的代码必须连续执行，而事件则是根据用户的操作随机触发的。

嗯，我觉得这一段比较难以解释，正所谓可意会而不可言传。希望读者自行体会一下。

有的朋友可能会说，这只是纯理论的区别而已，我怎么看起来最后还是一样？难道新的办法就不需要根据操作的情况设置各个命令按钮的 Enabled 状态了？请稍安勿躁，先看了余下的代码后我再做解释：

表单实例中处理信号的代码 Form.HandleSignal()

```
*********************************************************************
Lparameters cSignal

** 参数检查
IF pcount() = 0 or type("cSignal") # "C" or Empty(cSignal)
	Return
Endif

** 所有的耦合都集中在这里由表单来建立
** 因此，如果有需要改动的耦合，只要在这里改就一次搞定了
Do case
	Case cSignal == "ED_ADD"
		This.dataclass.ED_ADD()
	Case cSignal == "ED_SAVE"
"		this.dataclass.ED_SAVE()
	Case cSignal == "ED_UNDO"
		This.dataclass.ED_UNDO()
	Case cSignal == "ED_DELETE"
		This.dataclass.ED_DELETE()
	Case cSignal == "SETSTATUS"
		This.EditButtonGroup.SetStatus()
	Otherwise
```

```
            &&
ENDCASE
**********************************************************************
```

这里又出现了耦合，就是表单和 DataClass 对象之间的耦合、以及表单和命令按钮组 EditButtonGroup 之间的耦合。我要表达的意思就是：将耦合全部集中在表单与各个子对象之间，而各个子对象相互之间绝不建立耦合。这是因为：表单是永远存在的，而子对象是可能更换的。因此，子对象引用表单没有问题，而子对象要引用子对象，则应该通过表单来中转，这就是中介者模型的应用。

通过应用了中介者模型，当子对象被改动时，由于对子对象的引用都集中在表单上，我们只需要修改一下表单上对子对象的引用，就可以让程序继续顺畅的运行，而不会出现“找不到对象”的错误，更不需要找遍表单上每个子对象去一一修正各个子对象相互之间的引用。

理解了这一点，你就会知道，在数据对象类的各个方法（Ed_ADD()、Ed_Save()、Ed_Undo()）的代码中，若操作成功，则触发一个事件，该事件向表单发送一个信号“SETSTATUS”，否则，即简单返回。而表单根据该信号（见上面的代码中最后一个 Case）让命令按钮组 EditButtonGroup 设置各个子按钮的状态。

现在，回到前面我们提出的问题：面向过程跟面向对象的思路有什么区别呢？通过这两种不同的代码可以体会到：用面向过程的方法，就必然需要调用别的对象的方法，如此，就必然会出现耦合。而面向对象的方法则斩断了这种调用链，代之以触发一个事件来向表单发送一个信号的办法，如此，则可以取消可能会出现的耦合，而利用上中介者模型的好处。

总结

看这篇文章的我要收费！

呵呵，感觉这篇东西写下来，一下子把我一年来的心得体会全部都掏光了，免费送给大家的感觉真是…男人哭吧哭吧不是罪不是罪，我呜呜呜，再呜呜呜呜呜~~~~~~~~~~~~~~~

必须要说的是，这里面的不少东西也有 LQL 的心血，部分思想是在我们不断的讨论中总结出来的。在到温州之前，我基本没有研究过应用程序框架方面的内容，没有 LQL 的影响，这篇文章也不可能会出现…也许晚出现 N 年吧！大家为 LQL 鼓掌！谢谢！