FoxTalk

Август 2003
№ 8 (74)

русское издание

Solutions for Microsoft® FoxPro® and Visual FoxPro® Developers

Выявление «плохих» адресов электронной почты

WIN 7.0 6.0 5.0 DOWNLOAD

Уилл Хентцен (Whil Hentzen)

В этом месяце Уилл Хентцен обсуждает вторую половину действий с остатками массовой рассылки электронной почты — проверке, почему электронная почта возвращается к вам обратно, и то, как с ней следует поступать.

Как я уже упоминал ранее, когда вы выполняете какой-либо вид массовой рассылки электронной почты, то будете получать некоторое количество возвратов писем. Есть целый набор причин, почему это будет происходить. Наиболее частой причиной является то, что используемый вами адрес более не действует. Другими причинами могут быть неработающий сервер электронной почты, переполнение почтового ящика получателя или различные ошибки интернета (слишком много пересылок). Возможно даже, что адрес просто неправильный (пользователь ошибся, когда печатал свой адрес для вашей базы данных). А иногда это происходит потому, что электронные письма с вашего сервера отвергаются. Во всех этих случаях посланные вами электронные письма возвращаются к вам обратно на тот адрес, с которого они были посланы.

В течение последних нескольких месяцев я также получил много возвратов почты, вызванных действием вируса Klez и его клонов. Этот вирус довольно коварен, поскольку использует два подхода к распространению. Во-первых, как делает большинство вирусов, он использует преимущества незащищенной архитектуры Outlook, рассылая себя людям, внесенным в адресную книгу инфицированного компьютера. Во-вторых, в распространяемых им электронных письмах он подделывает заголовок «From» так, что кажется, что это электронное письмо пришло от кого-то еще из адресной книги, а не с инфицированного компьютера.

Другими словами, представьте себе, что инфицирован компьютер Эла, а в его адресной книге Outlook есть адреса Барбары и Карлы. Когда Klez атакует, он посылает электронное письмо Барбаре, но вместо адреса Эла использует для поля «From» адрес Карлы. Это особенно огорчительно, поскольку Барбара будет сердиться на Карлу из-за того, что получила от нее зараженное вирусом письмо, а вовсе не на Эла, являющегося действительным виновником.

Август 2003

8.0 7.0 6.0 5.0 3.0 2.x

материал имеет отношение к соответствующей версии

UNIX MAC DOS WIN

материал имеет отношение к соответствующей платформе

DOWNLOAD

исходные тексты программ можно скачать из Интернета

Причина, по которой мы получаем письма обратно, заключается в том, что люди иногда по ошибке помещают наш адрес электронной почты (books@hentzenwerke.com) в свои адресные книги. Затем, когда они подвергаются атаке вируса Klez, их компьютеры посылают электронные письма другим адресатам, но с указанием в поле «From» не действительного автора этого письма, а адреса books@hentzenwerke.com. Некоторые из адресов не работают по какой-либо причине, и эти письма возвращаются назад (или иногда отвергаются), а возвращенные сообщения посылаются на адрес отправителя — books@hentzenwerke.com.

В итоге можно сделать вывод, что когда имеешь дело с возвратом почты, то приходится разбирать груды мусора, представленного во множестве различных форматов.

Для получения возвращаемых писем и других откликов, вам нужно иметь отдельную учетную запись. Если же вы рассылаете огромное количество электронной корреспонденции, то, возможно, захотите иметь для этого более одной учетной записи. Для целей этой статьи я буду полагать, что вы создали только один электронный адрес исключительно для вашей массовой рассылки электронной почты и, таким образом, можете полностью отделить все возвращенные вам сообщения от других ваших адресов электронной почты. Как я уже говорил, вы будете по-прежнему получать массу мусора на эту учетную запись, приходящую в широком разнообразии форматов.

Почему так? Возвращаемые письма обрабатываются программой почтового сервера получателя, и различные почтовые сервера обрабатывают их поразному. Почтовый сервер — это просто программа, и сообщение, полученное сервером электронной почты и возвращаемое отправителю, является только частью этого программного кода. Поэтому письмо, возвращенное программой sendmail, будет отличаться от письма, возвращенного программой NTMail компании Gordano, или Lotus Notes.

Более того, сообщения от каждой программы могут отличаться в зависимости от того, как она была установлена и настроена. Письма, возвращенные поставщиком услуг интернета для Джо, пользующегося услугами SuperMail, могут отличаться от писем, возвращенных компанией MegaCom, также использующей SuperMail. И, в заключение, содержание и формат этих сообщений может отличаться в зависимости от причин, по которым письмо было отвергнуто. Несомненно, остается значительное число подобных вещей, есть так много способов сказать кому-либо: «Ваше сообщение было отвергнуто нашим сервером».

Для того, чтобы обработать каждый тип сообщений, необходимо знать, какие это форматы. Как вы определите их?

В предыдущей статье, посвященной удалению запросов, сначала я предполагал просмотреть мой Outlook PST, используя Office Automation, и сортировать сообщение за сообщением. Но теперь мы уже знаем, что это головная боль. Поэтому я экспортировал возвращенные сообщения в файл DBF и работал с ним (как было описано в статье прошлого месяца). После завершения, у меня были созданы файлы DBF и FPT, названные EMAILBAD.

Я проанализировал десятки тысяч возвратов писем в течение этого года и, к счастью, нашел значительное число сходств. По сути, есть две группы вернувшихся сообщений.

Первый тип содержит всю информацию, необходимую нам, в теле самого возвращенного сообщения. Вся необходимая информация содержится в заголовках From, Subject и Body для многих записей.

Второй тип не включает электронного адреса предполагаемого получателя, но содержит исходное сообщение электронной почты, как присоединенный файл. Этот случай вызывает больше затруднений, поэтому для целей этой статьи я собираюсь сосредоточиться только на тех электронных письмах, в которых я могу достигнуть цели, используя только содержание тела возвращаемых писем. Я нашел, что от 60 до 80 процентов возвращаемых назад писем мы можем успешно обработать, не обращаясь к присоединенному файлу.

Одинаковые строки Subject и, соответственно, схожие тела обнаруживаются снова и снова. Таким образом, общие для возвращенных писем строки “subject” и тела означают, что я могу анализировать эти сообщения автоматически, находить электронные адреса их получателей, искать их в моей базе данных CUST и устанавливать флажок, указывающий, что на этот адрес не следует больше посылать электронные письма (а также когда и почему этот флажок был установлен).

Я покопался в возвращенных сообщениях, датированных последними двумя годами, выполнил SELECT DISTINCT на столбце Subject и пришел к выводу, что обрабатывая только некоторые наиболее распространенные типы строк позаботишься о большей части возвращенных сообщений. Итак, как же извлечь из них адреса?

Идея заключается в том, чтобы просмотреть EMAILBAD.DBF, провести сортировку по строке “Subject” и изучить тело сообщения в поисках электронного адреса. После ручного поиска в сотнях из них, оказалось, что первый же электронный адрес в

теле практически всегда является адресом, на который электронное сообщение было послано. После того, как адрес определен, попытаемся сопоставить этот адрес с адресом в базе данных CUST.DBF. Если он будет найден, установим флажок в CUST.DBF, чтобы отметить этот адрес как «плохой», и обозначим причину: недоставленное, получатель неизвестен и т. д. Затем отметим флажком EMAILBAD, что адрес был найден (или не был). И в заключение, для полного завершения, архивируем файл PST на тот случай, если будет необходимо ссылаться на него снова.

Эта программа состоит из трех частей. Первая часть идентифицирует тип проверяемого сообщения и определяет некоторые параметры, указывающие на то, где может находиться адрес. Эти параметры включают в себя ту строку, в которой нужно искать (или в строке “subject”, или в теле), откуда начинать поиск (во многих случаях электронный адрес всегда находится после специфических текстовых строк, таких как «The following recipient was not found:» — «Следующий получатель не был найден:») и какие разграничители для адреса используются.

Например, сообщение со следующей строкой “subject”: «Mail System Error — Returned Mail» часто содержит в теле текст, похожий на следующий:

```
Each of the following recipients was rejected by a  remote
mail server. The reasons given by the server  are included
to help you determine why each recipient  was rejected.

   Recipient: <george.jungle@usa.net>
   Reason:    <george.jungle@usa.net>... User not known

Каждый из перечисленных ниже получателей был отвергнут уда-
ленным почтовым сервером. Причина, указанная сервером, по-
может вам определить, почему каждый из этих получателей
был отвергнут.

   Получатель: <george.jungle@usa.net>
   Причина:    <george.jungle@usa.net>... Пользователь
                                          неизвестен
```

Таким образом, строка, в которой будет производиться поиск, это весь блок текста. Нам нет необходимости начинать искать со слов «Each of the following» уже потому, что электронный адрес всегда находится после слова «Recipient:», поэтому эта строка является вторым параметром. Также электронный адрес всегда окружен разграничителями “<” и “>”. Значит именно эти четыре кусочка данных должны быть посланы во вторую часть программы, которая будет выбирать “george.jungle@usa.net” из тела этого сообщения.

Вторая часть программы находит адрес, заданный этими параметрами, и заносит полученное значение в неиспользуемое поле в E-MAILBAD.DBF (CCADDRESS — это такое же поле, как в программе EMAILREMOVE, рассмотренной нами в предыдущей статье).

И третья часть программы сканирует всю таблицу EMAILBAD в поисках адреса в CUST и устанавливает соответствующие флажки. Почему я не объединил вторую и третью части программы? Думаю, я мог бы, но поскольку выполнялось тестирование этих частей, мне необходимо было иметь промежуточные моментальные снимки данных для того, чтобы удостовериться, что извлекается нужная информация. Более эффективным было проверить таблицу EMAILBAD после сортировки адресов, но до поиска адресов в CUST.DBF.

Следующий код показывает операторы выбора для наиболее распространенных строк “subject”. Первый вариант не требует от нас сортировки тела сообщения, поскольку строка “subject” содержит электронный адрес, в доставке по которому было отказано.

```
case "@" $ m.lcSubject
  * не анализирует m.lcBody, поскольку достаточно
  * просмотреть строку “Subject”!

  m.lcStringToLookIn = m.lcSubject
  m.lcStringPre = ""
  m.lcStringDelim1 = "("
  m.lcStringDelim2 = ")"
  m.lcReason = "@ in subject line"
```

Второй вариант — это сообщения CompuServe, содержащие неизменный, легко опознаваемый текст как в строке “Subject”, так и в теле сообщения.

```
case m.lcSubject = "Undeliverable Message" ;
   and m.lcFromName = "CompuServe Postmaster"
  m.liPositionEnd = at("Sender: remove@hentzenwerke.com",;
    m.lcBody) + 32
  m.lcReason = "Undel/CompuServe PM/parsebody"
  m.lcStringToLookIn = substr(m.lcBody, m.liPositionEnd)
  m.lcStringPre = "for"
  m.lcStringDelim1 = "<"
  m.lcStringDelim2 = ">"
```

Третий вариант просто содержит электронный адрес в теле сообщения.

```
case m.lcSubject = "Returned mail:"
   m.lcReason = ["Returned mail:"]
   m.lcStringToLookIn = m.lcBody
   m.lcStringPre = ""
   m.lcStringDelim1 = "<"
   m.lcStringDelim2 = ">"
```

Четвертый вариант на самом деле служит для нескольких типов строки “Subject”, все из которых содержат строку «Undeliver» и строку «To:» в теле сообщения. Сообщения этих типов включают в себя «Undeliverable» и «Undelivered».

```
case m.lcSubject = "Undeliver:"
   m.lcStringPre = "To:"
   m.lcStringToLookIn = m.lcBody
   m.lcStringDelim1 = " "
   m.lcStringDelim2 = "chr(13) "
   m.lcReason = [Undeliverable]
```

Пятый вариант является необычным, поскольку строка “Subject” в нем начинается с буквы в нижнем

регистре, тогда как обычные строки “Subject” должны начинаться с прописной:

```
case m.lcSubject = "failure notice"
   m.lcStringPre = ""
   m.lcStringToLookIn = m.lcBody
   m.lcStringDelim1 = "<"
   m.lcStringDelim2 = ">"
   m.lcReason = [failure notice]
```

И в заключение, последний вариант состоит из двух проблем. В обоих случаях строка “Subject” читается так: «Mail System Error», но содержание тела сообщения форматировано одним из двух способов. Этот код обрабатывает их оба:

```
case m.lcSubject = "Mail System Error - Returned Mail"
  do case
  case "Recipient: <" $ m.lcbody and ;
     "Reason:    " $ m.lcbody
   m.lcReason = ["Recipient: <" $ m.lcbody and ;
     "Reason:    " $ m.lcbody]
   m.lcStringToLookIn = m.lcBody
   m.lcStringPre = "Recipient:"
   m.lcStringDelim1 = "<"
   m.lcStringDelim2 = ">"

  case "The following recipients did not receive ;
     this message:" $ m.lcbody
   m.lcReason = ["The following recipients did not ;
     receive this message:" $ m.lcbody]
   m.lcStringToLookIn = m.lcBody
   m.lcStringPre = "this message:"
   m.lcStringDelim1 = "<"
   m.lcStringDelim2 = ">"

  othe
   m.lcReason = "Mail System Error but otherwise CASE"
endcase
```

Теперь, когда параметры для электронного сообщения уже определены, я заношу это значение в неиспользуемое поле в EMAILBAD следующим образом:

```
replace BCCNAME with m.lcStringPre
replace BCCADDRESS with m.lcStringDelim1
replace BCCTYPE with m.lcStringDelim2
replace CCTYPE with m.lcReason
```

Итак, после того как я пропустил сообщение через код анализатора, я присвоил значения переменным m.lcReason, m.lcStringToLookIn, m.lcStringPre и получил два ограничителя. Я передам эти значения в код, обновляющий EMAILBAD, и вызову функцию, которая ищет электронный адрес и, если находит, возвращает его.

```
if m.lcReason <> "--"
  m.lcAddress = l_GetAddress(m.lcStringToLookIn, ;
     m.lcStringPre, m.lcStringDelim1, m.lcStringDelim2)

  if empty(m.lcAddress)
    * адрес не найден в cBody или cSubject
    * таблицы EMAILBADADDR.DBF
    replace CCADDRESS with "Not found"
  else
    replace CCADDRESS with m.lcAddress
  endif

 else
  replace CCADDRESS with "-- didn't test --"
endif
```

Я определил, что эти семь блоков анализатора позаботятся о 85 процентах всех возвращенных сообщений. Что касается прочего, то около 5 процентов сообщений нам вообще не нужно трогать – это сообщения о переполнении почтового ящика или о том, что система занята. Не нужно удалять эти сообщения, поскольку эти адреса в порядке – возможно что-то произошло из-за неполадок, которые будут исправлены в будущем и мы сможем опять рассылать по ним электронные сообщения.

Функция l_GetAddress является второй частью упомянутой мною программы. Это пользовательская функция, получающая четыре параметра и выполняющая поиск символа “@”. Она ищет строку m.lcStringToLookIn, начиная с места, в котором обнаружена m.lcStringPre. Эта функция также использует m.lcStringDelim1 и m.lcStringDelim2, чтобы знать, где осуществлять поиск.

Эта пользовательская функция не приводится в статье, но включена на дискету, сопровождающую журнал.

После того, как EMAILBAD заполнена адресами в поле CCADDRESS, я использовал процесс, похожий на программу REMOVE из прошлой статьи, для того, чтобы помечать таблицу CUST. Также эта программа добавляет статус success или failure для адресов из EMAILBAD, говорящий о том, найдены ли они в CUST. Заключительный шаг в этом процессе – это архивирование EMAILBAD и удаление электронных адресов из файла PST.

В отличие от процесса Remove, описанного в прошлой статье, этот процесс не будет таким совершенным, но он и должен быть таковым. Мы просто пытаемся сделать нашу массовую рассылку более эффективной, избавляясь от адресов, которые больше не работают. Нет необходимости проводить массу времени, разыскивая все эти электронные адреса до последнего. Я, например, рассчитываю затратить как можно меньшее время на то, чтобы получить этот результат.

Как и со многими другими вещами, это случай уменьшающейся отдачи — чем больше я хочу получить адресов, тем большее количество работы мне необходимо проделать. Если я упущу несколько, это не страшно. В отличие от проблемы с удалением запросов, возвращенная почта не причиняет беспокойства никому, и пересылка от 2 до 4 процентов дополнительной почты не является большой проблемой для нашего почтового сервера. Хотя ваш расчет, конечно, может отличаться от моего.

❑❑❑

Модель поведения

Нэнси Фолсом (Nancy Folsom)

В своей последней статье («Каким способом переданы данные?») Нэнси Фолсом показала пример классов данных, использующих как собственные данные FoxPro, так и ADO. Она применяла эти классы в сочетании с бизнес-объектами так, что источник данных являлся прозрачным не только для этой формы (уровень интерфейса пользователя), но также и для уровня бизнес-логики. На этот раз она рассматривает код для объектов на форме в примере из прошлой статьи, чтобы посмотреть, как пользовательский интерфейс взаимодействует с уровнем бизнес-логики.

Моделирование и разработка классов бизнес-логики и данных является непростой задачей и заслуживает того, чтобы широко обсуждаться в статьях и на конференциях. Однако я была разочарована. Эта тема практически не освещается, несмотря на то, что зачастую разработка удобных компонентов пользовательского интерфейса для работы с бизнес-объектами может оказаться нетривиальной. Есть такое мнение, что классы пользовательского интерфейса не моделируются — за исключением вариантов использования (use cases). Я понимаю эту точку зрения, но когда приходит время сесть и создать форму для ввода данных любой степени сложности, между пользовательским интерфейсом и бизнес-объектами обнаруживается множество неясных границ, поэтому так просто размыть их, в результате же, обязанности элементов приложения оказываются инкапсулированными некачественно.

Ретрансляция новостей

Пользовательский интерфейс ввода данных будет иметь различные обязанности, такие как отображение информации, предоставление некоторых средств для внесения, сохранения и отмены изменений и перемещения по записям. Какие элементы управления можно использовать для этих задач — достаточно очевидно. Кнопки обеспечивают способ для пользователей выполнять действия, поля ввода (TextBox) — механизм для ввода данных и так далее. Их координация внутри пользовательского интерфейса и с уровнем бизнес-логики может оказаться сложной проблемой. Требования для формы ввода данных SPOOPPaymentsEx7.SCX, показанной на рис. 1, следующие:

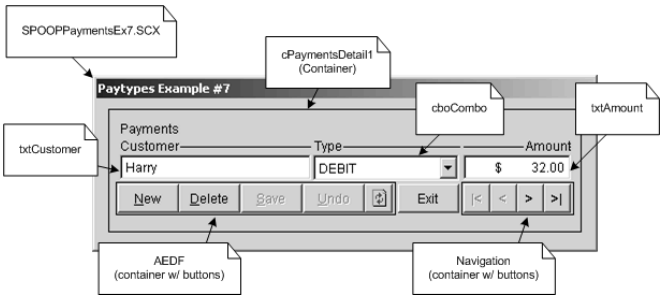

Рис. 1. Платежная форма (перед тем, как были сделаны какие-либо изменения), указывающая на классификацию объектов.

- Когда я изменяю вид платежа, количество или имя заказчика, я хочу включить кнопки Save и Undo.
- Эти кнопки должны быть включены только в том случае, если некоторые изменения еще не завершены.
- Если я откладываю платеж (например, перехожу к следующему) и у меня есть незавершенные изменения, я хочу сохранить их.

В конечном счете, событие в каком-либо одном объекте (например, в TextBox), будет иметь логическое воздействие на другой объект (скажем, CommandButton). Если придерживаться правила помогать пользователю при выборе, то некоторые кнопки должны быть доступными или недоступными в зависимости от текущего контекста. Например, кнопка Save знает, что имя заказчика изменилось. Цель, однако, заключается в том, чтобы поддерживать функциональность, заключенную в CommandButton, независимой от функциональности, заключенной в TextBox. Почему? Вполне возможно, что этот порядок вещей изменится в будущем. В итоге мне, возможно, потребуется иметь таблицу заказчика и тогда бы я заменила TextBox на ComboBox, перечисляющий заказчиков. Изменение или даже удаление одного из этих объектов не должно воздействовать на кнопку Save и наоборот. Решение заключается в том, чтобы объекты рассылали сообщения, когда что-либо случается. Тогда (подобно слушателям радио), другие объекты могут решить, стоит ли им реагиро-

вать на это событие или даже хотят ли они вообще слушать это. Кнопку Delete не должно заботить изменение имени заказчика, а вот для кнопки Save это важное событие. Почему? Да потому, что на самом деле я разговариваю с моим компьютером. Серьезно, если я рассматриваю эти элементы в качестве активных участников, это помогает мне представить структуру в целом. «Заботиться» — означает просто то, что состояние объекта может нуждаться в изменении, поскольку что-то произошло где-то в другом месте.

«Модифицирующие» кнопки (new, delete, save и undo) заботятся о том, были ли сохранены измененные данные. Кнопки для перемещения заботятся о расположении текущего платежа относительно всех других платежей. Какого рода события должны интересовать объекты? Кнопку Save не должно заботить ни то, какая информация изменилась, ни то, где произошло это изменение, в TextBox или EditBox. Кнопку Next не должно заботить то, что текущий отображаемый вид платежа — это “CASH”. Все, что объект должен знать, — это общая (абстрактная) информация, такая как: «что-то изменилось» или «текущая запись меняется». Объекты не должны зависеть от получения конкретных сообщений в определенном порядке (любые подобные «вещатели» должны быть удалены из контейнера). Передающие объекты должны быть независимы от любых слушающих объектов.

Итак, вот те классы, которые я выбрала для примера, и вот почему. «Платежи» являются реальными логическими сущностями, составленными из разделенных физически элементов данных, находящихся в одной или нескольких таблицах. В данном примере: «Заказчик» + «Тип» + «Количество» = «Платеж». Поскольку один платеж состоит из нескольких частей, контейнер является естественным элементом пользовательского интерфейса для его отображения. Всякий раз, когда я хочу создать графический интерфейс для сущности, я начинаю с класса контейнеров, даже для набора кнопок. На самом деле, контейнер хорошо соответствует моделированию понятия составного класса.

Вложенные объекты могут рассылать сообщения своим родительским объектам. Те, в свою очередь, могут пересылать эти сообщения любым другим объектам (подписчикам), которые могут слушать или отвергать это сообщение. Давайте рассмотрим пример. Когда я меняю имя заказчика и затем выхожу с помощью клавиши табуляции из поля заказчика (как это показано на рис. 2), существует несколько вещей, которые, возможно, должны произойти:

Рис. 2. Форма ввода платежа после изменения имени заказчика и выхода с помощью клавиши табуляции из поля Customer.

- Потребуется проверка достоверности входных данных.
- Потребуется проверка достоверности изменения бизнес-объектом.
- Потребуется проверка достоверности изменения объектом по работе с данными.
- Если все прошло проверку, тогда кнопки Save и Undo должны быть активированы.

Поскольку контейнер является владельцем этой сущности (платеж = контейнер), и сущности управляются с использованием бизнес-объектов, этот контейнер владеет бизнес-объектом. Проверка достоверности является обязанностью бизнес-объекта. Поэтому мой объект TextBox для ввода данных заказчика знает о бизнес-объекте Payments. Таким образом, когда контейнер Payments инициализирован, он создает соответствующий бизнес-объект и затем прикрепляет его ко всем членам контейнера, осведомленным об этом бизнес-объекте. Что означает «быть осведомленным»? Обычно я использую метод, именуемый AttachBizObj. Если объект имеет этот метод, он будет вызван. Однако если такого метода нет, то программа продолжает работу, игнорируя этот объект. Другими словами, программа приспособлена к обоим видам объектов. Путем использования метода вместо прямого присваивания этого объекта свойству члена, я вправе менять структуру этих объектов, включая добавление массива бизнес-объектов.

```
*!* aEntityUI.Init()
 IF !EMPTY( THIS.BizObjClass )
      LOCAL lni, lo
   *!* Создает бизнес-объект
      lo = NEWOBJECT(THIS.BizObjClass,THIS.BizObjClassLib)
   *!* Прикрепляет ссылку на вложенные объекты,
   *!* если требуется
      FOR lni = 1 TO THIS.CONTROLCOUNT
        IF PEMSTATUS( THIS.CONTROLS(lni),'AttachBizObj',5)
                      THIS.CONTROLS(lni).AttachBizObj(lo)
        ENDIF
      NEXT lni
   *!* Сохраняет эту ссылку на свойство
   *!* контейнера сущности.
      THIS.ADDPROPERTY('BizObj',lo)
      RELEASE lo
 ENDIF
```

Обратите внимание на то, что код написан в абстрактном, высокоуровневом классе. Он на два уровня удален от контейнера payments (см. диаграмму класса на рис. 1). В качестве примера, ниже приведен метод aTextBox.AttachBizObj:

```
*!* aTextBox.AttachBizObj()
 LPARAMETERS toBizObj
 THIS.ADDPROPERTY('bizObj',toBizObj)
```

После того, как имя заказчика было изменено, и я ушла при помощи клавиши табуляции с этого поля, сработает метод Valid(). Он может ссылаться на бизнес-объект контейнера и не знать никаких деталей о том, что делает проверка достоверности (в данном случае немного):

```
*!* aTextBox.Valid()
 IF THIS.OLDVAL # THIS.VALUE
        THIS.BizObj.VALIDATE(THIS)
        THIS.PARENT.DIRTYBUFFER = .T.
 ENDIF
```

(OldVal — это просто индивидуальный набор свойств в методе GotFocus, позволяющий мне легко понять, что пользователь сделает действительные изменения). Метод Valid моего базового класса ComboBox выполняет ту же самую работу. Обратите внимание на присваивание значения для DirtyBuffer. Примененная мною уловка заключается в использовании метода Assign для свойства DirtyBuffer_Assign так, что когда состояние буфера меняется, объект может делать что-либо с информацией и, если это необходимо, транслировать это изменение. В этом случае метод DirtyBuffer_Assign() контейнера payments выглядит так:

```
*!* cPaymentDetail.DirtyBuffer_Assign():
 LPARAMETERS vNewVal
 LOCAL lo, lni, lnCount
 lnCount = THIS.CONTROLCOUNT
 THIS.DirtyBuffer = vNewVal
 FOR lni = 1 TO lnCount
        lo = THIS.CONTROLS(lni)
        IF PEMSTATUS(lo, 'DirtyBuffer', 5)
                lo.DirtyBuffer = vNewVal
        ENDIF
 NEXT lni
```

Когда флажок DirtyBuffer платежа изменится, все вложенные объекты, которые хотели бы знать об этом, будут знать. Именно так кнопка Save узнает о том, что имя заказчика изменилось. Лучшим выбором, чем прямое назначение DirtyBuffer, будет вызов метода Set. Например:

```
Set("DirtyBuffer", vNewVal)
```

Этот способ более гибок и допускает изменение механизмов объектов без воздействия на среду. В любом случае, поскольку используется тот же основной класс контейнеров, что и cPaymentDetail, мы имеем одинаковый DirtyBuffer_Assign и будем передавать сообщение вниз вложенным в него объектам (включая кнопку Save) абсолютно тем же способом. Конечный результат показан на рис. 3.

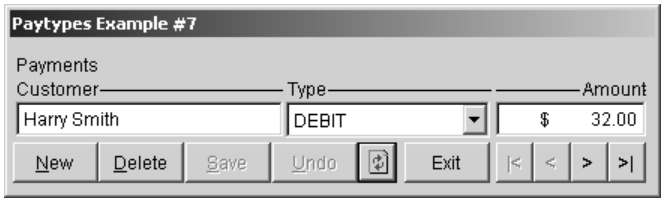

Рис. 3. Объект Payment после сохранения изменений в имени заказчика.

С другой стороны, когда пользователь щелкает кнопку Save, метод Click() базового класса вызывает срабатывание специального метода Do(), имеющего (в случае кнопки Save), следующий код:

```
*!* btnSave.Do()
 LOCAL loError
 loError = THIS.PARENT.SAVE()
 THIS.PARENT.DirtyBuffer = !ISNULL( loError )
```

И опять объект (кнопка) посылает сообщение своим родительским объектам для установки флажка DirtyBuffer. DirtyBuffer_Assign() срабатывает, и кнопка Undo, например, может самоотключиться после того, когда изменения уже сохранены. На рис. 4, 5 изображены диаграммы последовательности, иллюстрирующие эти взаимодействия объектов. Рис. 6 показывает диаграмму соответствия классов для этих объектов, использованных в диаграмме последовательности.

Перемещение

Переходя к кнопкам для перемещения, мы видим типичные, только до некоторой степени неудобные, кнопки для прокрутки платежных записей. Я использую слово «запись» не совсем правильно. Помните, что вы не можете обращаться напрямую к данным в пользовательском интерфейсе. В данном случае, «запись» просто означает следующий платеж, как пользователь увидит его. Как и в случае с контейнером, содержащим кнопки New, Delete, Save и Undo, этот навигационный контейнер также воздействует на данные. Поскольку весь доступ осуществляется через этот бизнес-класс, контейнер будет нуждаться в ссылке на бизнес-класс платежной сущности. К счастью, это не является проблемой, поскольку он использует тот же самый базовый класс базового контейнера (посмотрите еще раз на рис. 6).

Например, когда пользователь щелкает кнопку Last — это вызывает метод GoLast родительского объекта:

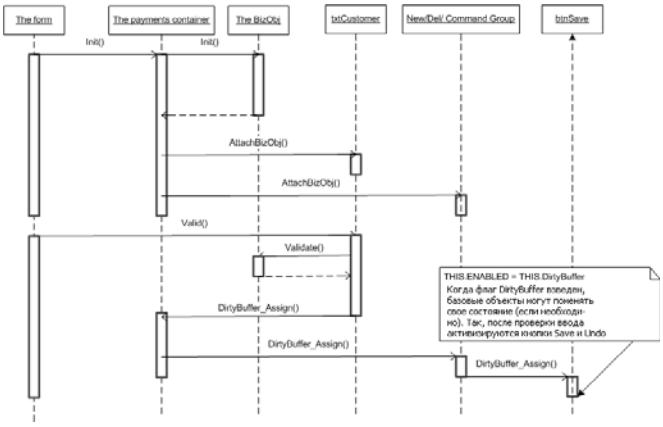

Рис. 4. Последовательность событий, происходящих при инициализации бизнес-объекта и изменении имени заказчика.

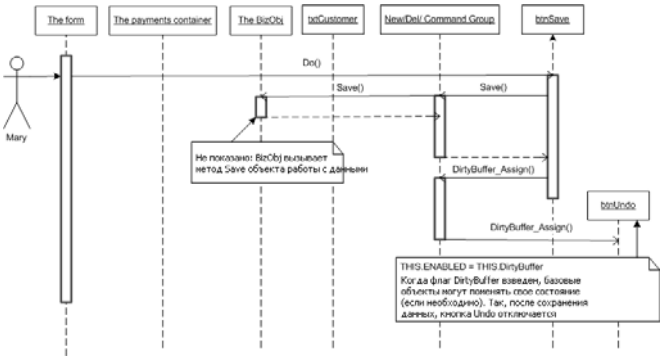

Рис. 5. Последовательность событий в операции сохранения.

```
*!* btnLast.Do()
THIS.BizObj.SAVE()
THIS.BizObj.MoveLast()
THISFORM.REFRESH()
```

Обратите внимание, что метод Save вызывается первым (это тот же самый метод, который, в конечном итоге, вызывает кнопка Save). В данном примере я делаю предположение, которое обычно не стала бы делать на практике: пользователь хочет сохранить любые незавершенные изменения. Обычно я могу вежливо осведомиться об этом. Далее объекту BizObj велено исполнить метод MoveLast. Пока нет ни одного объекта, который бы озаботился тем, что обозначает термин «последний» или как подойти к этому делу. Пользовательский интерфейс не заботится ни об одной из этих деталей. Это неважно для него, но метод BizObj.MoveLast просто повторяет указание объекту данных (ретранслирует это сообщение), а затем обновляет данные, чтобы они были доступны в пользовательском интерфейсе.

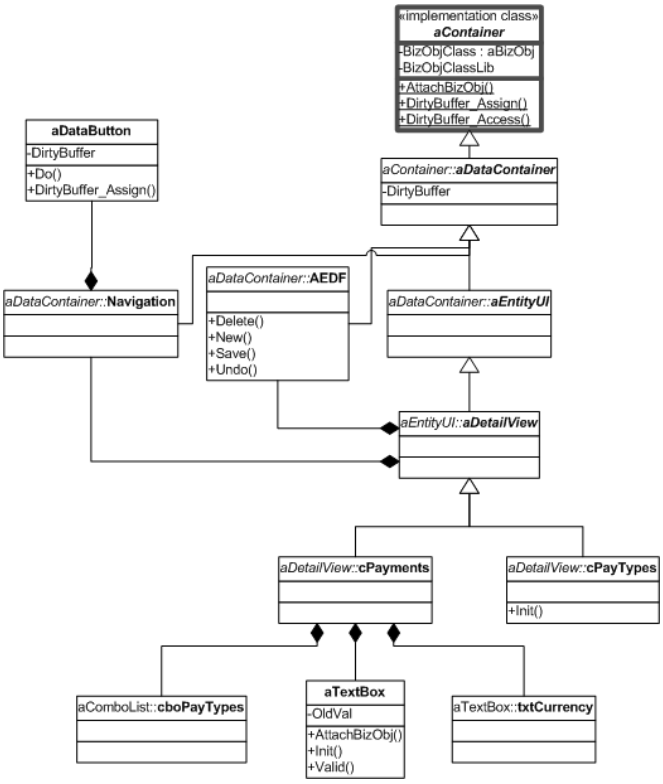

Рис. 6. Диаграмма класса объектов пользовательского интерфейса, используемых в диаграммах последовательности на рисунках 4 и 5.

В следующий раз

Когда я вижу представления систем моделирования, то зачастую бываю разочарована, поскольку пользовательский интерфейс не включается в обсуждение моделирования. Делается акцент на бизнесе и данных. Существует мнение, что пользовательский интерфейс не должен моделироваться за исключением вариантов использования. Несмотря на то, что варианты использования чрезвычайно полезны, я все же мечтаю о другом шаге. В следующем месяце я рассмотрю, как можно моделировать пользовательский интерфейс для программы импорта и сравню его с вариантом использования, а также объясню, почему я думаю, что пользовательский интерфейс часто пропускается в моделировании.

Нэнси Фолсом (Nancy Folsom) занимается разработкой xBASE-приложений обработки бизнес-данных как для частных, так и для правительственных заказчиков, начиная с 1989 года. Она публикует свои статьи в журналах CoDe Magazine (издательство EPS Publishing) и Virtual Fox User Group Newsletter и является автором книги Debugging Visual FoxPro Applications (издательство Hentzenwerke Publishing). nancy@mvps.org.

❑❑❑

Функции INLIST и ASCAN — делаем пару зарубок на память

Прадип Ачарья (Pradip Acharya)

WIN 7.0 6.0 DOWNLOAD

В то время как VFP изначально является независящим от регистра языком, такие функции, как INLIST и ASCAN, являются исключениями из правила. Более того, критерий сопоставления зависит от настройки EXACT, а также может возникнуть несоответствие типов данных. Надлежащее использование этих функций требует утомительного и повторяющегося кодирования. В этой статье Прадип Ачарья демонстрирует, что использование расширенных версий этих программ может привести к замечательному упрощению кода и улучшению достоверности. Также описаны расширенный Equality Checker и более удобная ADEL.

Кто побаивается функции INLIST? Конечно, я. Я опасаюсь использования INLIST и ASCAN из-за встроенных в них ловушек. Во-первых, INLIST подвержена проблеме несоответствия типов данных (в то время как ASCAN — нет). Во-вторых, для символьных данных есть две неопределенности, о которых следует позаботиться, а именно: чувствительность к регистру и настройка EXACT. При обычном использовании я не беспокоюсь о регистре (например, для переменной или имени файла) и не забочусь о частичном совпадении слева. Если я ищу только один APPLE, orange или Peach, я не буду писать 10 строчек кода или делать вставку UPPER(ALLTRIM(m.MyVar)) для того, чтобы вызвать INLIST без опасений.

Возьмем программу Genmenu.prg. Код страдает точно таким же недостатком. Я проверил мой громоздкий код и сделал вывод, что должен существовать простой способ для вызова INLIST, ASCAN или ADEL. Эта проблема также включает в себя определение равенства двух переменных. Тот факт, что 123 не равно “ABC” интуитивно ясен. Почему же тогда я получаю ошибку?

Так же, для практических целей, PrePaid эквивалентно PREPAID. Очень часто я хочу удалить один из элементов выпадающего списка. Я не могу вызвать функцию ADEL, потому что не знаю его положения в массиве. Мне необходимо вызвать сначала ASCAN, и тут начинается цирк. После того, как я преодолею этот барьер, я остаюсь с избыточным значением .F. в конце, означающим, что я должен вручную изменить размерность массива.

Решение всех этих задач заключено в использовании четырех расширенных функций — xINLIST, xASCAN, xADEL и xEQ. Я был удивлен тем упрощением кода, к которому привела очистка моего приложения при помощи этих новых функций. Но это еще не все. Еще одно преимущество заключается в большей надежности. Например, представьте себе, что имя процедуры хранится в поле memo. Если в нем есть лишние символы, UPPER(ALLTRIM()) без посторонней помощи не сможет осуществить подгонку, необходимую перед проверкой равенства. Но нам не стоит так беспокоиться, если мы будем использовать любую из этих новых функций. Исходный код для этих четырех функций доступен на дискете. Для того, чтобы установить их для тестирования, разархивируйте их и введите:

```
SET PROCEDURE TO XINLIST ADDITIVE
```

Расширенная INLIST

На практике невозможно выполнить следующее при помощи INLIST:

```
OnOff = xINLIST(m.MyVar, "Yes", 1, .T., "Y", "ON", ;
   "T", ".t. ", "True")
```

Для xINLIST, как видите, это пустяк. Что еще можно сказать?

Типичный вызов INLIST использует код, подобный следующему (см. Genmenu.prg):

```
PRIVATE SaveExact
 SaveExact = SET("EXACT")
 SET EXACT ON
 If INLIST(UPPER(ALLTRIM(m.MyVar1)), ;
   UPPER(ALLTRIM(m.ListItem1)), ;
   UPPER(ALLTRIM(m.ListItem2)))
     ...
 endif
 If m.SaveExact == "OFF"
   SET EXACT OFF
 endif
```

Теперь предположим, что m.ListItem1 или m.ListItem2 хранит что-то иное, чем символьную строку. Вам потребуется еще 10 строк кода, чтобы справиться с проблемой несоответствия типов данных. Вы можете заменить целый блок кода следующим:

```
If xINLIST(m.MyVar1, m.ListItem1, m.ListItem2)
 ...
endif
```

В качестве дополнительного приза, вам не надо заботиться о несоответствии типов данных, о наличии

любого из символов перехода на новую строку и так далее в любой из строк, преобразующихся с высокой надежностью.

xINLIST вызывается с теми же аргументами, что и обычная INLIST, и не зависит от регистра, типа данных и настройки Set Exact. Строки символов автоматически подгоняются. Чтобы выполнить сравнение, длины должны быть одинаковыми. Вы можете вызывать до 27 элементов списка (см. листинг 1).

Листинг 1. Функция xINLIST.

```
FUNCTION xINLIST
 LPARAMETERS ;
     p1, p2, p3, p4, p5, p6, p7, p8, p9,  ;
     p10,p11,p12,p13,p14,p15,p16,p17,p18, ;
     p19,p20,p21,p22,p23,p24,p25,p26,p27

 local npar, i, temp, item, NullFlag, p1type
 npar = PARAMETERS()
 if ISNULL( m.p1 )
     return NULL
 endif

 p1type = VARTYPE(m.p1)
 if m.p1type $ "CM" && логика обработки строк
     temp = UPPER(xTRIM( m.p1 ))
     for i = 2 to m.npar
         item = EVAL("m.p" + ALLTRIM(STR(m.i)))
         DO CASE
         case ISNULL( m.item )
             NullFlag = NULL

         case (VARTYPE( m.item ) $ "CM") and ;
          (UPPER(xTRIM( m.item )) == m.temp)
             return

         ENDCASE
     endfor
 else              && Другие типы данных
     for i = 2 to m.npar
         item = EVAL("m.p" + ALLTRIM(STR(m.i, 2)))
         DO CASE
         case ISNULL( m.item )
             NullFlag = NULL

         case (VARTYPE( m.item ) == m.p1type) and ;
          (m.item = m.p1)
             return

         ENDCASE
     endfor
 endif
 return m.NullFlag
```

Проверка равенства

Если i1 и i2 являются целыми числами, а c1 и c2 строками символов, тогда действием, противоположным (i1 = i2), является (i1 != i2). Для данных типа Character действием, противоположным (c1 == c2), является не (c1 != c2), а !(c1 == c2). Почему? Потому что (c1 != c2) зависит от настройки EXACT. Один только этот подвох может привести к ошибочному коду. Чтобы проиллюстрировать это утверждение, приведу следующий код:

```
SET EXACT OFF
?"xxx" == "xx" && отображает .F.
?"xxx" != "xx" && также отображает .F.
```

Расширенный Equality Checker (xEQ) обходит эти проблемы и не порождает никаких ошибок несоответствия данных, если аргументами являются, например, 123 и “ABC”. xEQ является частным случаем функции xINLIST, только с двумя аргументами. Обратите внимание, что и xINLIST и xEQ может возвращать значение NULL, если какой-либо из аргументов имеет значение NULL. Интересно, что xEQ() будет возвращать значение .T., поскольку обоими предполагаемыми аргументами являются .F.. И xEQ(1) будет возвращать .F., поскольку предполагаемым вторым аргументом является .F., не равный 1 (см. листинг 2).

Мощность функции xEQ иллюстрируется следующим примером:

```
x="APPLES"
 y="apples" + CHR(13)
 ?UPPER(ALLTRIM(m.x)) == ;
  UPPER(ALLTRIM(m.y))  && неправильный ответ .F.
 ?xEQ(m.x, m.y)        && правильный ответ .T.
 ?xEQ(123, "ABC")      && правильный ответ .F.
```

Листинг 2. Функция xEQ.

```
FUNCTION xEQ  && возвращает .T., .F. или Null
 LPARAMETERS p1arg, p2arg
 if ISNULL( m.p1arg ) or ISNULL( m.p2arg )
     return NULL
 endif
 local temp
 temp = VARTYPE( m.p1arg )
 if not (m.temp == VARTYPE( m.p2arg ))
     return .f.
 endif
 if m.temp $ "CM"
     return (UPPER(xTRIM(m.p1arg)) == ;
             UPPER(xTRIM(m.p2arg)))
 endif
 return (m.p1arg = m.p2arg)
```

Расширенная ASCAN

В отличие от xINLIST, xASCAN содержит дополнительный пятый аргумент, не представленный в обычной ASCAN. Пятый аргумент, p5atc (см. листинг 3), означает, что при определении символьного типа сравнения (character type match) применяется функциональная возможность функции ATC, независящей от регистра. Например, следующий код будет определять, существует ли совпадение, если PrePaid будет найден где-либо в одном из элементов массива:

```
xASCAN(@m.MyArray, "PREPAID",,,"Slide")
```

xASCAN также не зависит от типа данных, регистра и установки EXACT. Обратите внимание на разницу в передаче ссылки на массив между ASCAN и xASCAN:

```
ASCAN(MyArray, "LookFor")
xASCAN(@m.MyArray, "LookFor")
```

И хотя мы сосредоточены на типе данных String, xASCAN в равной степени хорошо работает со

всеми типами данных. Также здесь отсутствует проверка для значений NULL (см. листинг 3).

Листинг 3. Функция xASCAN.

```
FUNCTION xASCAN      && нет проверки на значения NULL
 LPARAMETERS p1array, p2what, p3i1, p4ntot, p5atc
 external array p1array

 local i, i1, i2, vType, LookFor, temp
 i2 = ALEN( p1array )
 DO CASE      && устанавливает начальный и конечный элементы
 case empty( m.p3i1 )
     i1 = 1

 case m.p3i1 > m.i2
     return 0

 other
     i1 = m.p3i1

 ENDCASE
 if not empty( m.p4ntot )
     temp = m.i1 + m.p4ntot - 1
     if m.temp < m.i2
         i2 = m.temp
     endif
 endif

 vType = VARTYPE( m.p2what )
 if m.vType $ "CM"
     LookFor = UPPER(xTRIM( m.p2what ))
 endif
 DO CASE
 case not (m.vType $ "CM")
     for i = m.i1 to m.i2
         if (VARTYPE(p1array( m.i )) == m.vType) and ;
            (p1array( m.i ) = m.p2what)
             return m.i
         endif
     endfor

 case empty( LookFor )
     for i = m.i1 to m.i2
         if (VARTYPE(p1array( m.i )) $ "CM") and ;
             empty(xTRIM(p1array( m.i )))
             return m.i
         endif
     endfor

 case empty( m.p5atc )
     for i = m.i1 to m.i2
         if (VARTYPE(p1array( m.i )) $ "CM") and ;
        (UPPER(xTRIM(p1array( m.i ))) == m.LookFor)
             return m.i
         endif
     endfor

 other
     for i = m.i1 to m.i2
         if (VARTYPE(p1array( m.i )) $ "CM") and ;
            (ATC(m.LookFor, p1array( m.i )) > 0)
             return m.i
         endif
     endfor

 ENDCASE
 return 0
```

Ошибка передачи по ссылке в VFP

При передаче любых массивов в любую функцию, вы можете столкнуться с ошибкой низкоуровневой адресации в VFP (включая версию 7). Это проявляется также и при вызове функции xASCAN. Ошибка показана в листинге 4.

Листинг 4. Ошибка передачи по ссылке в VFP.

```
dimension MyVar( 3 )
create table temp (MyVar L)
Proc1( @MyVar ) && выдает сообщение о мнимой ошибке:
** Function argument value, type or count is invalid
Proc Proc1
lpar aList
extern array aList
return
```

Косвенный префикс “&” обрабатывается VFP корректно, но ссылочный префикс “@” — нет. VFP знает, что “&” может быть применен только к переменным, но не к полям таблицы. Для префикса “@”, применяются точно такие же критерии. Напротив, VFP дает сбой при наличии поля таблицы с таким же именем.

Решение заключается в том, что нужно всегда присоединять префикс “m.” к имени переменной. В предыдущем примере код не вызовет ошибки, если вы выполните следующее:

```
Proc1( @m.MyVar )
```

Однако проблема на этом не заканчивается. По существу, вы не можете передать один единственный элемент массива по ссылке в любое время без риска. Например, следующий синтаксис отвергается, поскольку недопустим:

```
Proc1( @m.MyVar( 2 ))
```

Единственный безопасный обходной путь для передачи одиночного элемента по ссылке состоит в следующем:

```
Local temp
Temp = MyVar( 2 )
Proc1( @m.temp)
MyVar( 2 ) = m.temp
```

Подобные низкоуровневые ошибки — и не только одна эта — требуют дополнительного кодирования и приводят к внезапным необъяснимым отказам в распределенных приложениях VFP.

Расширенная ADEL

xADEL не обладает большим сходством со стандартной ADEL. Например, второй аргумент является не порядковым номером элемента, а критерием совпадения для расположения удаляемого элемента. Символьный тип сравнения не зависит от типа данных, регистра или настройки EXACT. Третий аргумент, p3atc (см. листинг 5), означает, что когда устанавливается символьный тип сравнения, то применяется функциональная возможность ATC по поиску вхождения строки без учета регистра. Например, этот код будет удалять первый найденный элемент, содержащий где-либо в себе строку “Deliver”:

```
xADEL(@m.MyArray, "DELIVER", "Slide")
```

Обратите внимание на разницу в передаче ссылки на массив между ADEL и xADEL:

```
ADEL(MyArray, 5)
xADEL(@m.MyArray, "LookFor")
```

Хотя мы сосредоточили свое внимание на типе данных String, xADEL одинаково хорошо работает со всеми типами данных. Здесь нет проверки значений NULL. xADEL автоматически убирает замыкающее значение .F. после удаления и возвращает .T., если совпадение было найдено и удаление было выполнено, в противном случае возвращает .F.. xADEL идеально подходит для удаления одного элемента из выпадающего списка (см. листинг 5).

Листинг 5. Функция xADEL.

```
FUNCTION xADEL  && Поиск, уничтожение и восстановление

 lpar ap1, p2LookFor, p3atc
 external array ap1
 local i
 i = xASCAN(@m.ap1, m.p2LookFor,,,p3atc)
 if empty( m.i )
     return .f.
 endif

 ADEL(ap1, m.i)
 if ALEN( ap1 ) > 1
     dime ap1(ALEN( ap1 ) - 1)
 else
     dime ap1( 1 )
     ap1( 1 ) = .f.
 endif
 return
```

Заключение

Расширенные функции xINLIST и xASCAN позволяют избежать волнения и неповоротливости, связанной с применением стандартных функций INLIST и ASCAN. Использование этих функций приводит к поразительному упрощению кода и повышает достоверность. xADEL автоматически ищет удаляемые элементы в массиве и удаляет замыкающие .F. после удаления этих элементов. xEQ определяет равенство двух переменных независимо от типа данных, регистра, настройки EXACT или наличия посторонних символов. Работая вместе, эти функции могут в значительной степени уменьшить и упростить написание кода.

Прадип Ачарья (Pradip Acharya) является разработчиком и консультантом, работающим в Торонто (Канада). До поступления в 1992 году в Gencom Software, Inc., он работал для компании British Petroleum. Прадип занимается низкоуровневой функциональностью в VFP и специализируется в написании библиотек DLL и FLL на языке C для расширения возможностей VFP. В последние годы он поставлял важные системы для таких компаний, как Steelcase Canada и Billiton Metals. Его электронный адрес: pacharya@look.ca.

❑❑❑

2D/3D-формы и элементы интерфейса

Предраг Боснич (Predrag Bosnic)

Заинтересованы ли вы в приложениях, использующих пользовательский интерфейс, выполненный в 2D/3D-стиле? Предраг Боснич показывает, как это сделать.

В сентябре 1992 года я получил работу по контракту в маркетинговой компании. Я разрабатывал программу управления контактами с добавочным финансовым модулем и встроенными коммуникационными возможностями (автоматический факс). Как и многие другие, я чувствовал наступающую экспансию Windows и хотел сделать это приложение под Windows, а не под DOS.

В то же время мой работодатель хотел чего-нибудь нового и волнующего, он хотел, чтобы продукт соответствовал ракетно-космическим технологиям, используемым его компанией. Ранее я имел опыт работы с менфреймами и персональными компьютерами. Конечно, единственным правильным решением являлось использование персональных компьютеров, но сначала я должен был найти базу данных, которая работала бы под Windows. Я уже имел опыт работы с такими продуктами, как dBase, Clipper и Fox. В то время VB еще не работал с данными; Access еще не родился; Fox, dBase и Clipper имели только DOS-версии. Кроме того, вся моя работа должна была быть закончена за три месяца. Единственным решением, которое я нашел, была dbFast — продукт компании Computer Associates — база данных, совместимая с xBase и работающая под Windows.

И я решил взять программу dbFast (поскольку она совместима с базами данных dBase), изучить ее по ходу дела и, возможно, у меня был шанс закончить работу в назначенное время.

На следующее утро коробка с dbFast была на моем столе. У меня не хватило терпения просмотреть файл readme. Ясно, что install.exe должен был находиться на первом диске. После нескольких минут программа была установлена (исполняемый файл был меньше 500Кб, если мне не изменяет память). Я щелкнул пиктограмму dbFast, чтобы запустить программу, и новый мир распахнулся передо мной. Теперь у меня была совместимая с dBase база данных, и я мог создавать мое первое приложение для работы с базами данных в Windows.

Я хотел создать «стильные» трехмерные формы с эффектными трехмерными элементами интерфейса, но был очень разочарован. Все доступные элементы были двухмерными, не считая командной кнопки. Моя форма была очень плоской и очень отличалась от форм Windows. Возможно, это и не было настолько плохо, но я хотел трехмерных впечатлений и ощущений.

Однако через несколько месяцев приложение было готово, и все были счастливы за исключением меня. Я продолжал хотеть моих трехмерных форм и элементов интерфейса.

В феврале 1993 года, на показе Windows в Лондоне, Microsoft представила Access 1.0 с предварительной ценой 100 фунтов. Я оказался одним из тех счастливчиков, кто смог унести с собой с показа большую коробку с руководством и дискетами с программным обеспечением. Я не пытался спрятать эту коробку, полагая, что каждый завидовал мне, потому что я обладал совершенно новой базой данных Windows. Что это было за время – покупаешь программный продукт и получаешь все эти руководства и дискеты! Конечно, я продолжал изучать, как использовать это новое программное обеспечение, но я, наконец, мог создавать мои трехмерные формы.

В марте 1993 года в моих руках оказалась FoxPro 2.5 для Windows. Можете вы в это поверить? Совместимая с dBase база данных под Windows — и она может создавать трехмерные формы. Мне казалось, что это сон!

Спустя почти десять лет, за окном 21 век, Лондон, dbFast более не существует (насколько я знаю); MS Access вышел уже в версии XP; Fox в версии 7; VB стал VB.NET. Но, по некоторым причинам, мои формы продолжают оставаться двухмерными. Нет, это не ошибка. Все элементы управления остаются двухмерными, формы — двухмерными, MS Office XP — двухмерный. Что происходит? Все просто, на пользовательский интерфейс тоже есть мода. А она ходит по кругу. Моя дочь носит джинсы 1960 года. И мои формы продолжают оставаться двухмерными, если я хочу оставаться «стильным» (если хочу выжить).

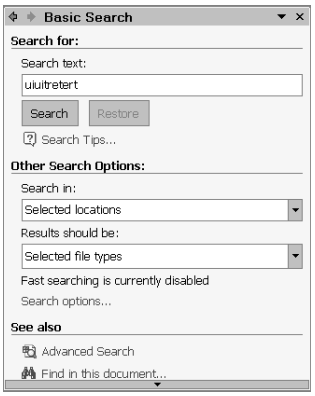

Рис. 3. MS Word XP, панель задач: простота — вот имя для игры с этой красавицей.

Однако сегодня все меняется. И если вы согласны с тем, что пользовательский интерфейс очень важен, а используемый нами инструмент (Visual FoxPro) обладает огромными возможностями, тогда он позволяет нам, как разработчикам, делать наши приложения более современными и привлекательными. При этом я говорю исключительно о пользовательском интерфейсе, а не о функциональных возможностях. Благодаря технологии (в первую очередь ООП), встроенной в Visual FoxPro, я смело могу сказать: «Только небо является пределом!».

2D/3D-формы и элементы интерфейса (прототип)

Посмотрите на пакет MS Office XP и, в особенности, Word XP. Он опять плоский! Окно панели задач является плоским и содержит, в большинстве случаев, проские элементы интерфейса (иногда один или два элемента внутри оказываются трехмерными, например, панель Reveal Formatting или Translate — похоже на то, что у Microsoft проблемы с элементом ListBox). Я хотел бы видеть нечто похожее на это в Fox — и опять я говорю только о пользовательском интерфейсе, а не о функциональности (см. рис. 3).

Я хочу построить прототип, чтобы посмотреть, могу ли я создать 2D/3D-формы и элементы интерфейса в Visual FoxPro. Также хотелось бы, чтобы мой код был настолько гибким, насколько это возможно. Пользователь должен иметь возможность выбора стиля интерфейса приложения или изменять этот стиль в любой момент. Это означает, что для одного и того же приложения я могу иметь как двухмерные, так и трехмерные настройки в одно и то же время. Это может работать также как цветовая схема Windows (будем надеяться, если трехмерность снова станет модным в течение 10 лет, пользователи моих приложений не будут испытывать никаких проблем).

Прежде всего для моей формы (когда активен 2D-стиль) нужен светло-серый цвет. Как найти значение RGB (Red/Green/Blue) для этого цвета? Выполните шаги, используемые мною:

- 1. Захватите изображение на экране (используя клавишу PrintScreen клавиатуры).
- 2. Запустите приложение MS Paint.
- 3. Вставьте изображение из буфера обмена.
- 4. Выберите инструмент Select the Pick Color на панели инструментов и щелкните область, цвет которой вам нужен.
- 5. На цветовой панели инструментов вы можете видеть, что основной цвет автоматически станет тем, который вы щелкнули (выбрали).
- 6. Теперь щелкните меню Colors и выберите опцию Edit Colors. Появится окно Edit Color.
- 7. Щелкните кнопку Define Custom Colors. Окно Edit Color расширится.
- 8. На правой его стороне вы можете прочитать значение RGB для выбранного цвета.

В моем случае я получил следующие настройки: RGB(246,246,246). А что насчет элементов управления? На этом этапе я не вижу никаких проблем. Каждый элемент управления имеет свойство SpecialEffect. Это свойство может принимать следующие значения:

- 0 – 3D (трехмерный).
- 1 – Plain (2D) (плоский).
- 2 – Hot Tracking (изменение внешнего вида элемента при перемещении поверх его курсора мыши).

В зависимости от значения этого свойства, Fox будет отображать любой элемент интерфейса трехмерным, плоским или с эффектом Hot Tracking.

К сожалению, не все элементы интерфейса поддерживают значение Plain и эффект Hot Tracking. Наибольшие же проблемы возникают у элементов Grid и PageFrame.

Я могу создать свойство на уровне приложения для поддержки значения настройки SpecialEffect или, чтобы сделать то же самое, просто использовать глобальную переменную gnAppStyle.

Метод Init этого элемента содержит следующий код:

```
IF type('gnAppStyle') = 'N' and gnAppStyle >=0 ;
   and gnAppStyle < 3
    this.SpecialEffect = gnAppStyle
ENDIF
```

Еще раз взгляните на изображение панели Word XP (рис. 3). Вы можете заметить, что границы всех элементов интерфейса имеют темно-серый цвет. Элементы интерфейса Fox, однако, имеют черные границы, если их свойству SpecialEffect задано значение “Plain”. Однако я могу изменить это просто:

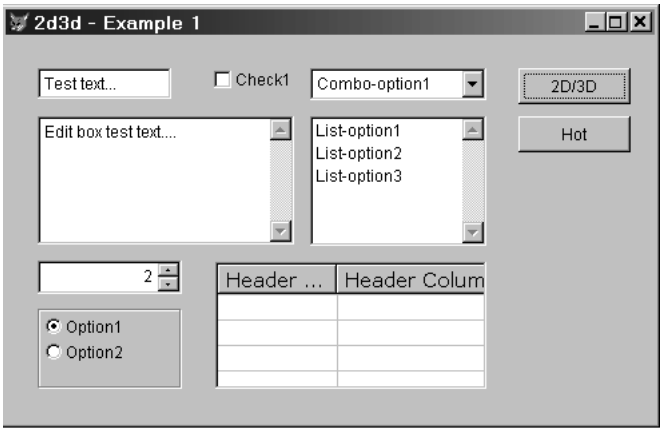

Рис. 4. Форма в 3D-стиле.

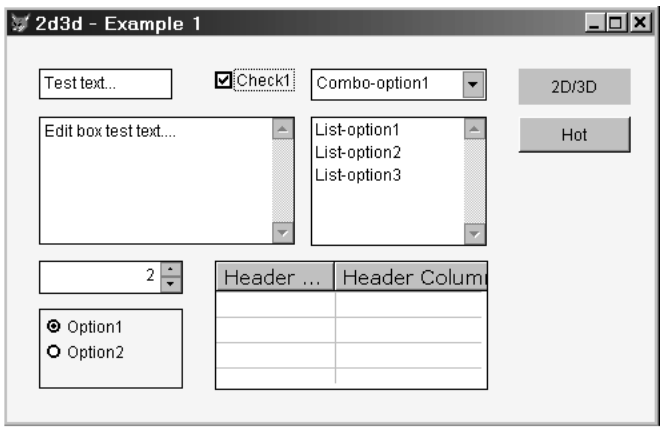

Рис. 5. Форма в 2D-стиле.

```
IF type('gnAppStyle') = 'N' and gnAppStyle >=0 and;
  gnAppStyle < 3
   this.SpecialEffect = gnAppStyle
   IF gnAppStyle = 1 OR gnAppStyle = 2
     this.BorderColor = rgb(128,128,128)
   endif
ENDIF
```

Если я помещу этот кусочек кода в метод Init всех элементов интерфейса моей тестовой формы, тогда мой образец будет выглядеть так, как это показано на рис. 4 и рис. 5.

Важно напомнить, что в этом прототипе я не использовал элемент интерфейса Grig (Fox), а использовал ListView (MS), поддерживающий трехмерный и плоский стили.

С моей точки зрения, этот небольшой прототип доказывает, что я могу построить форму, которая ме-

няет свой внешний облик «на лету». Я учитываю проблемы, возникающие с элементами управления Grid и PageFrame, но все остальное выглядит прекрасно. Теперь я могу создать мое окончательное решение.

Окончательное решение

Для того, чтобы поддержать стандартные цветовые схемы Windows, границы всех элементов интерфейса должны иметь соответствующий цвет. Как найти нужное цветовое число? Вот благоприятная возможность представить вам небольшую, но полезную функцию Windows API — GetSysColor. Эта функция принимает только один параметр: изображение элемента, чей цвет необходимо определить. В загружаемом файле вы найдете форму SysColors.scx, демонстрирующую использование этой функции. Элементы изображения, которые мне интересны, это:

- 3 = COLOR_INACTIVECAPTION
- 4 = COLOR_MENU
- 16 = COLOR_BTNSHADOW
- 15 = COLOR_BTNFACE
- 20 = COLOR_BTNHIGHLIGHT

Теперь я могу создать библиотеку класса, содержащую важные видимые элементы интерфейса. Для выполнения этой работы я использовал встраиваемое дополнительное средство Tamar Granor Class Browser: NewLib (см. www.universalthread.com, в разделе Downloads). Эта утилита создает копию любой библиотеки класса. В данном случае я хочу копировать библиотеку Visual FoxPro _Base.vcx в новую библиотеку, называемую 2d__base.vcx.

Следующий код есть в методе Init каждого видимого элемента интерфейса:

```
IF type('gnAppStyle') = 'N' and gnAppStyle >=0 and;
   gnAppStyle < 3
   this.SpecialEffect = gnAppStyle
   IF PemStatus(this,'BorderColor',5)
     IF gnAppStyle = 1 OR gnAppStyle = 2
       this.BorderColor = getSysColor(16)
     endif
   endif
ENDIF
```

Элемент интерфейса Grid не имеет свойства SpecialEffect, но вы знаете, что он является трехмерным элементом (см. рис. 6). Однако я хочу попробовать применить некоторые идеи и, возможно, сымитировать плоский внешний вид. Прежде всего посмотрите на элемент интерфейса Grid.

Заголовки, метки записей и полосы прокрутки являются трехмерными. Если я смогу изменить их, тогда я найду решение. Метки записей практически не используются, и я могу просто отключить их.

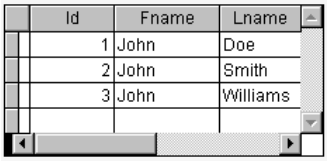

Рис. 6. Элемент интерфейса Grid в VFP не имеет свойства SpecialEffect.

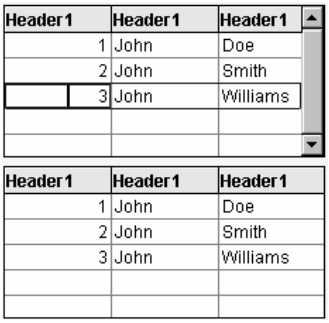

Рис. 7. Плоский Grid в VFP.

Для прокрутки записей я могу использовать клавиатуру, отказавшись от полос прокрутки, или, по меньшей мере, отключить горизонтальную полосу. Объект header имеет свойство BackColor. Я могу установить это свойство и изменить внешний вид. Если я буду использовать «классическую» цветовую схему Windows, тогда хорошим выбором будет цвет RGB (230,230,230), а для всех других цветовых схем я могу использовать GetSysColor(20). Последняя настройка выполняется для свойства GridLineColor. Я использую GetSysColor(4) для того, чтобы «оживить» Grid. После завершения этих действий моя сетка выглядит так, как показано на рис. 7.

Ниже приведен код из метода Init этого элемента Grid:

```
DoDefault()
 If Type('gnAppStyle') = 'N' And gnAppStyle >= 0 ;
   And gnAppStyle < 3
   If gnAppStyle = 0    && 3D
     For Each oColumn In This.Columns
       oColumn.Header1.BackColor = GetSysColor(15)
       oColumn.Header1.FontBold = .F.
     Next
   Else
     For Each oColumn In This.Columns
       LOCAL jnColor as Integer
       jnColor = GetSysColor(20)
       If jnColor = 16777215
         oColumn.Header1.BackColor = ;
         Rgb(230,230,230)
       Else
         oColumn.Header1.BackColor = ;
         GetSysColor(20)
       Endif
       oColumn.Header1.FontBold = .T.
     Next
     This.GridLineColor = GetSysColor(4)
   Endif
Endif
```

Последней проблемой, которую мне пришлось решать, был элемент интерфейса PageFrame. Он имеет свойство SpecialEffect, но это свойство применяется только в том случае, если отключено свойство Tabs. В этом частном случае цель заключалась в том, чтобы сделать BackColor каждой вкладки таким же, как BackColor формы. Если вы помните, я использовал цвет RGB (246,246,246) для BackColor формы.

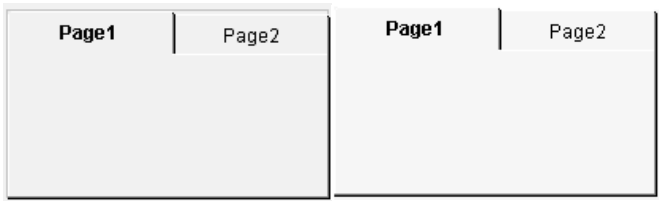

Рис. 8. Плоский элемент PageFrame в VFP.

Рис. 9. Плоский элемент PageFrame с добавлением линий и рамок.

Если я применю этот цвет к элементу PageFrame, я получу результат, показанный на рис. 8.

Думаю, он выглядит достаточно плоско. Единственная проблема заключается в верхней и левой линии рамки. Я мог бы обойтись без этих линий, но у меня есть забавная идея. Представьте себе шаблонный объект, помещенный за PageFrame, с цветом границ установленным в GetSysColor(16). Верхняя и левая координаты этого шаблона на единицу меньше, чем у PageFrame. Теперь у меня есть граница вокруг элемента PageFrame. Для того, чтобы управлять таким поведением автоматически, элементу PageFrame необходимо определяемое пользователем свойство cBackBorder. Это свойство содержит имя шаблонного объекта, используемого для получения эффекта рамки.

Метод Init элемента PageFrame содержит следующий код, и результат его выполнения показан на рис. 9.

```
DoDefault()
 If Type('gnAppStyle') = 'N' And gnAppStyle >= 0 ;
   And gnAppStyle < 3
   LOCAL jcName as String, jn1 as Integer
   If gnAppStyle = 0       && 3D
     For jn1 = 1 To This.PageCount
       This.Pages(jn1).BackColor = GetSysColor(15)
     Next
     jcName = This.cBackBorder
     This.Parent.&jcName..Visible = .F.
   Else
     For jn1 = 1 To This.PageCount
       This.Pages(jn1).BackColor =Rgb(246,246,246)
     Next
     jcName = This.cBackBorder
     With This.Parent.&jcName
       .Left = This.Left - 1
       .Top  = This.Top - 1
       .Width = This.Width - 1
       .Height = This.Height
       .BorderColor = GetSysColor(15)
       .BackStyle = 0
       .Visible = .T.
       .ZOrder(1)
     Endwith
   Endif
Endif
```

И, в заключение, я сделал изображения формы Test_2dClass.scx для того, чтобы проверить 2D/3D-свойства. Как видно из рис. 10 и рис. 11, элемент PageFrame имеет границу.

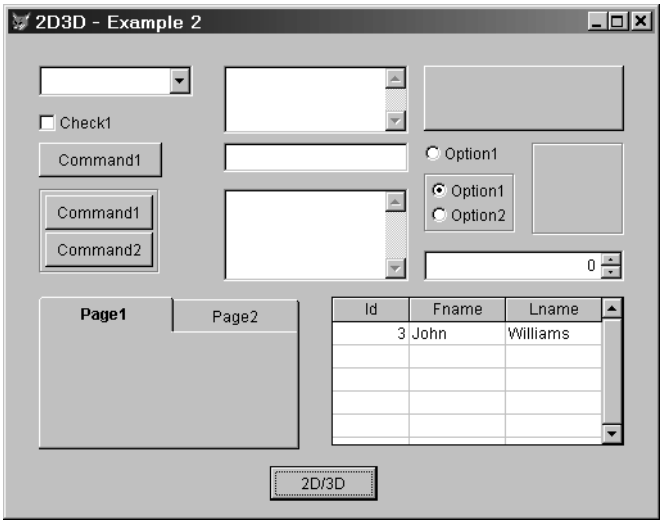

Рис. 10. Проверяем форму — 3D-стиль.

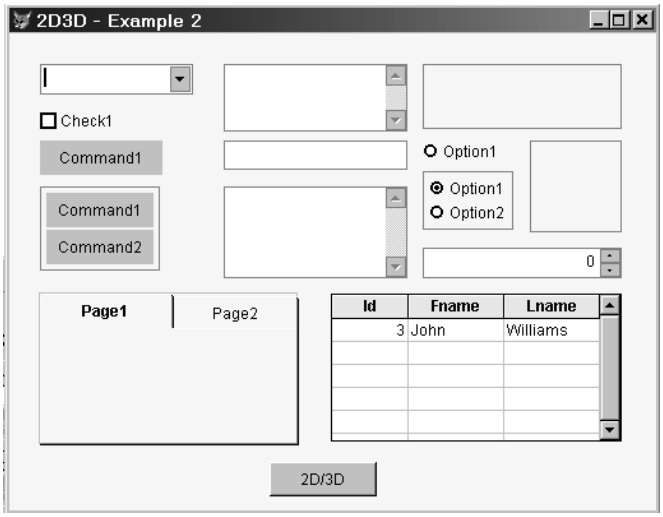

Рис. 11. Проверяем форму — 2D-стиль.

Заключение

В результате я получил библиотеку класса со всеми основными видимыми элементами интерфейса и теперь могу просто переключаться между 2D и 3D-стилями. Если вы проанализируете выполненную мною работу, то увидите, что работы по написанию кода было совсем немного, и решение выглядит довольно простым. Это то, что я называю «мощью Visual FoxPro».

❑❑❑

Поймай меня, если сможешь

Дуг Хенниг (Doug Hennig)

WIN 8.0 DOWNLOAD

VFP 8.0 имеет структурированную обработку ошибок, содержащую в себе новые структуры TRY … CATCH … FINALLY … ENDTRY. Эти новая мощная возможность обеспечивает трехуровневую обработку ошибок и позволяет вам исключить большое количество кода, связанного с пересылкой и обработкой информации об ошибках. В этом месяце Дуг Хенниг обсуждает структурированную обработку ошибок и показывает, как она вписывается в общую стратегию обработки ошибок.

Добавив метод Error к объектам, VFP 3 значительно улучшил возможности обработки ошибок в FoxPro. Это позволило объектам инкапсулировать свою собственную обработку ошибок и не полагаться на глобальные программы обработки ошибок. Однако одним из недостатков размещения кода в методе Error вашего объекта является то, что он замещает команду ON ERROR. И это понятно, поскольку в противном случае будет нарушаться инкапсуляция. Давайте посмотрим, в чем заключается проблема. Пусть объект, содержащий код в своем методе Error, вызывает процедурный код (такой как PRG) или метод другого объекта, не имеющего кода в своем методе Error. Если ошибка произойдет в этом вызываемом коде, то сработает метод Error вызывающего объекта, даже если вызываемый код имеет установленный локальный обработчик ON ERROR. С этим механизмом связаны еще две проблемы:

- Многие типы ошибок могут быть заблаговременно предупреждены, такие, например, как попытка открыть таблицу, монопольно используемую кем-либо еще. Однако, поскольку обработчик ON ERROR, установленный в вызываемой процедуре, не срабатывает, эта процедура не получает возможности обработать собственные ошибки.
- Поскольку вызывающий объект не знает с каким типом ошибок может столкнуться вызываемая процедура, то непонятно, как он будет обрабатывать их (за исключением типового метода: регистрация ошибки, отображение типового сообщения для пользователя, выход и так далее).

Ясно, что нам необходим лучший механизм. К счастью, VFP 8 дает нам более подходящий инструмент: структурированную обработку ошибок (structured error handling).

Структурированная обработка ошибок

В C++ структурированная обработка ошибок существует уже давно. .NET добавляет структурированную обработку ошибок в языки, в которых прежде ее не было, например в VB.NET. Так что это за штука — структурированная обработка ошибок? Структурированная обработка ошибок означает, что код исполняется специальными блоками или структурами и, если любая ошибка случится в этом коде, другая часть этой структуры обрабатывает ее.

VFP 8 осуществляет структурированную обработку ошибок следующим способом:

- Структура TRY ... ENDTRY позволяет выполнить код, возможно вызывающий ошибку, и обрабатывает ее в этой структуре. При этом все другие обработки ошибок отвергаются.
- Новая команда THROW позволяет пересылать ошибки в обработчик ошибок более высокого уровня.
- Новый базовый класс Exception обеспечивает объектно-ориентированный способ передачи информации об ошибках.

Давайте рассмотрим эти улучшения.

Пытаемся еще раз

Новая структура TRY ... ENDTRY является ключевой в структурированной обработке ошибок. Вот ее синтаксис:

```
try
   [ TryCommands ]
 [ catch [ to VarName ] [ when lExpression ]
   [ CatchCommands ] ]
   [ exit ]
   [ throw [ uExpression ] ]
 [ catch [ to VarName ] [ when lExpression ]
   [ CatchCommands ] ]
   [ exit ]
   [ throw [ uExpression ] ]
 [ ... (additional catch blocks) ]
 [ finally
   [ FinallyCommands ] ]
endtry
```

VFP будет пытаться выполнить команды в блоке TryCommands. Если ошибок не возникает, то исполняется код в необязательном блоке FINALLY (если он есть), а затем продолжается выполнение кода,

следующего после ENDTRY. Если возникнет любая ошибка в блоке TRY, VFP немедленно выходит из этого блока и начинает выполнение операторов CATCH.

Если VarName включена в оператор CATCH, VFP создает объект Exception, заполняет его свойства информацией об этой ошибке и помещает ссылку на этот объект в переменную VarName. VarName может быть только обычной переменной, но не свойством объекта. Если вы предварительно объявили эту переменную, она будет иметь любую область видимости, заданную вами (например LOCAL); если нет, она будет определена как PRIVATE. Мы рассмотрим базовый класс Exception позднее.

Когда используется необязательная конструкция WHEN, операторы CATCH могут действовать как операторы CASE. Выражение WHEN в этой конструкции должно возвращать логическое значение, так чтобы VFP могла решать, что делать. Если это выражение .T., то код в этом блоке оператора CATCH выполняется, если .F., то VFP переходит к следующему оператору CATCH. Этот процесс продолжается до тех пор, пока выражение WHEN оператора CATCH 1) не вернет значение .T.; 2) не будет достигнут оператор CATCH без конструкции WHEN или 3) больше не останется операторов CATCH (мы обсудим этот случай позднее). Обычно выражение WHEN проверяет свойства объекта Exception (такие как ErrorNo, содержащие номер этой ошибки).

Когда VFP находит оператор CATCH, выполняются команды в этом блоке. После того, как этот блок будет выполнен, выполняется код в необязательном блоке FINALLY (если он есть), и выполнение продолжится для кода, следующего за ENDTRY.

Следующий пример, взятый из SimpleTry.prg (включенный на дискету), показывает как оцениваются операторы CATCH и что структура TRY замещает установку ON ERROR:

```
on error llError = .T.
 llError = .F.

 try
   wait window xxx
 catch to loException when loException.ErrorNo = 1
   wait window 'Error #1'
 catch to loException when loException.ErrorNo = 2
   wait window 'Error #2'
 catch to loException
   lnError = loException.ErrorNo
   messagebox('Error #' + transform(lnError) + chr(13) + ;
     'Message: ' + loException.Message)
 finally
   messagebox('Finally')
 endtry

 on error
 messagebox('ON ERROR ' + iif(llError, 'caught ', ;
   'did not catch ') + 'the error')
```

Если VFP не находит оператор CATCH, имеет место необрабатываемое исключение (ошибка 2059). Этого не нужно допускать по ряду причин, главная из которых заключается в том, что проблема, вызвавшая исходную ошибку, не обрабатывается, поскольку теперь мы имеем еще большую путаницу.

Далее приведен пример того, что случается, когда вы имеете необрабатываемое исключение (взятое из UnhandledException.prg). Когда вы запустите этот код, то увидите сообщение о том, что ошибка является необрабатываемым исключением, вместо сообщения об истинной причине: «Переменная XXX не существует»:

```
on error do ErrHandler with error(), program(), lineno()
 try
   wait window xxx
 catch to loException when loException.ErrorNo = 1
   wait window 'Error #1'
 catch to loException when loException.ErrorNo = 2
   wait window 'Error #2'
 finally
   messagebox('Finally')
 endtry
 on error

 procedure ErrHandler(tnError, tcMethod, tnLine)
 local laError[1]
 aerror(laError)
 messagebox('Error #' + transform(tnError) + ;
   ' occurred in line ' + transform(tnLine) + ' of ' + ;
   tcMethod + chr(13) + 'Message: ' + message() + ;
   chr(13) + 'Code: ' + message(1))
```

Вы не можете вернуться назад

Важное различие между структурированной обработкой ошибок и другими типами обработки ошибок в VFP заключается в том, что вы не можете вернуться назад к коду, вызвавшему ошибку. В методе Error или в процедуре ON ERROR есть только несколько способов продолжения:

- RETURN (или неявный возврат без использования оператора RETURN) возвращает к строке кода, следующей за той, что вызвала ошибку. Это обычно приводит к появлению других ошибок, поскольку код, вызвавший первую ошибку, не выполнил своей задачи (инициализация переменной или открытие таблицы).
- RETRY возвращает к строке кода, вызвавшей ошибку. До тех пор, пока проблема не будет исправлена каким-либо магическим способом, это, очевидно, будет вызывать ту же проблему снова.
- QUIT завершает работу приложения.
- RETURN TO возвращает к процедуре из стека вызовов, такой как та, что содержится в операторе READ EVENTS. Это очень полезно, если вы хотите оставаться в приложении, а не возвращаться к процедуре, вызвавшей ошибку. Конечно, это не означает, что все обстоит благополучно, но это

часто позволяет пользователю предпринять еще какие-то действия, если ошибка не является катастрофической (например, простая проблема борьбы за обладание ресурсами, в ходе попытки войти в форму).

При использовании структуры TRY в случае ошибки блок TRY покидается, и вы не можете вернуться в него. Если вы используете RETURN или RETRY в блоке CATCH (фактически, если вы используете их где-либо в этой структуре), то вызовете появление ошибки 2060. Вы можете, конечно, использовать QUIT для завершения работы этого приложения.

В заключение мы можем убрать за собой

Мне потребовалось не так много времени, чтобы разобраться, зачем необходима конструкция FINALLY. Кроме того, код, следующий за оператором ENDTRY, исполняется независимо от того, имела место ошибка или нет. Оказывается, в действительности это не всегда верно; как вы увидите при обсуждении команды THROW, ошибки могут «проявиться» только в следующем, более высоком уровне обработчика ошибок, и вы, возможно, не получите возврата от этого обработчика. Это означает, что вы не можете гарантировать, что код, следующий за ENDTRY, будет выполнен. Однако вы можете гарантировать, что код в блоке FINALLY всегда исполняется (ну, почти всегда: если обработчик ошибок COM-объекта вызывает COMRETURNERROR(), выполнение немедленно возвращается на COM-клиента).

Есть пример, демонстрирующий это (UsingFinally.prg). Вызов функции ProcessData включен в структуру TRY. ProcessData сама имеет структуру TRY, но она только обрабатывает ошибку и не способна монопольно открывать таблицу, поэтому ошибка WAIT WINDOW XXX не может быть захвачена. Как следствие, эта ошибка «проявится» во внешней структуре TRY в главной процедуре и поэтому код, следующий за ENDTRY в ProcessData, не будет исполняться. Закомментируем блок FINALLY в ProcessData и запустим этот код. Вы увидите, что таблица Customers остается открытой после его исполнения. Выведем блок FINALLY из комментария и запустим его снова — вы увидите, что на этот раз таблица Customers будет закрыта, так что этот код должным образом «убирает» за собой.

```
try
   do ProcessData
 catch to loException
   lnError = loException.ErrorNo
   messagebox('Error #' + transform(lnError) + ' occurred')
 endtry
 if used('customer')
   messagebox('Customer table is still open')
else
   messagebox('Customer table was closed')
endif used('customer')
close databases all

function ProcessData
try
   use (_samples + 'data\customer') exclusive
* выполняет некоторую обработку
   wait window xxx
* Обрабатываем ситуацию,
* что мы не можем открыть таблицу монопольно.
catch to loException when loException.ErrorNo = 1705
* Код-"мусорщик". Закомментируйте/снимите комментарий
* с блока, чтобы увидеть разницу.
finally
   if used('customer')
     messagebox('Closing customer table in FINALLY...')
     use
   endif used('customer')
endtry

* Теперь происходит очистка. Этот код не может быть
* исполнен, поскольку ошибка "проявилась".
if used('customer')
   messagebox('Closing customer table after ENDTRY...')
   use
endif used('customer')
```

Объект Exception

VFP 8 включает в себя новый базовый класс Exception, позволяющий предоставлять объектно-ориентированные методы передачи информации об ошибках. Как вы видели ранее, объекты Exception создаются при использовании конструкции TO VarName в командах CATCH. Также они создаются при использовании команды THROW, которую мы обсудим позднее.

Кроме обычных свойств, методов и событий (Init, Destroy, BaseClass, AddProperty и им подобных), Exception обладает набором свойств (показанных в таблице 1), содержащих информацию об ошибках. Все они доступны для чтения-записи в период исполнения.

Перебрось его мне

Последней частью структурированной обработки ошибок является новая команда THROW. Она напоминает команду ERROR, поскольку также вызывает пересылку ошибки к обработчику ошибок, но она работает несколько по-другому. Вот синтаксис этой команды:

```
throw [ uExpression ]
```

uExpression может быть всем, чем угодно (по-вашему желанию), например, сообщением, численным значением или объектом Exception.

Если значение uExpression определено, THROW создает объект Exception, устанавливает его свойство ErrorNo в значение 2071, Message — в “User Thrown Error”, а UserValue получает значение из uExpression. Если uExpression не было определено, используется исходный объект Exception (тот, что

Таблица 1. Свойства нового базового класса Exception.

Свойство	Тип данных	Схожие функции	Описание
Details	Character	SYS(2018)	Дополнительная информация об ошибке (такая, как имя переменной или несуществующего файла). NULL, если не применимо.
ErrorNo	Numeric	ERROR()	Номер ошибки.
LineContents	Character	MESSAGE(1)	Строка в коде, вызывающая ошибку.
LineNo	Numeric	LINENO()	Номер строки.
Message	Character	MESSAGE()	Сообщение об ошибке.
Procedure	Character	PROGRAM()	Процедуры и методы, в которых произошла ошибка.
StackLevel	Numeric	ASTACKINFO()	Уровень стека вызовов для процедуры.
UserValue	Variant	Аналога нет	Выражение, определенное в операторе THROW.

был создан, когда произошла ошибка) при условии, что он существует; если нет, создается новый объект Exception. В любом случае объект Exception пересылается на следующий, более высокий уровень обработчика ошибок (являющегося, как правило, структурой TRY, включающей в себя структуру TRY, из которой была вызвана команда THROW). Вот пример, взятый из TestThrow.prg:

```
try
   try
     wait window xxx
   catch to loException when loException.ErrorNo = 1
     wait window 'Error #1'
   catch to loException when loException.ErrorNo = 2
     wait window 'Error #2'
   catch to loException
     throw loException
   endtry
 catch to loException
   lnError   = loException.ErrorNo
   lcMessage = loException.Message
   messagebox('Error #' + transform(lnError) + chr(13) + ;
     'Message: ' + lcMessage, 0, 'Thrown Exception')
   lnError   = loException.UserValueErrorNo
   lcMessage = loException.UserValueMessage
   messagebox('Error #' + transform(lnError) + chr(13) + ;
     'Message: ' + lcMessage, 0, 'Original Exception')
 endtry

 * Теперь сделаем то же самое, но будем использовать
 * THROW без каких-либо выражений
 try
   try
     wait window xxx
   catch to loException when loException.ErrorNo = 1
     wait window 'Error #1'
   catch to loException when loException.ErrorNo = 2
     wait window 'Error #2'
   catch to loException
     throw
   endtry
 catch to loException
   lnError   = loException.ErrorNo
   lcMessage = loException.Message
   messagebox('Error #' + transform(lnError) + chr(13) + ;
     'Message: ' + lcMessage, 0, 'Thrown Exception')
 endtry
```

Вы, возможно, решите, что команда THROW loException в этом коде «перебрасывает» loException на следующий, более высокий уровень обработчика ошибок. Однако это не тот случай. THROW создает новый объект Exception и «перебрасывает» его, помещая loException в свойство UserValue этого нового объекта. Таким образом, код в предшествующей внешней структуре TRY показывает, что полученный им объект Exception является «переброшенной» пользователем ошибкой. Чтобы получить информацию об исходной ошибке, вам потребуется получить свойства того объекта Exception, на который ссылается свойство UserValue.

Второй пример в предыдущем коде показывает, что THROW сама будет повторно перебрасывать объект Exception, если он будет иметься в наличии. В этом случае, вы имеете точно такой же объект, так что внешний обработчик ошибок не будет иметь ссылок на свойство UserValue.

Вы можете использовать оператор THROW вне структуры TRY, но тогда он не будет обладать никакими дополнительными преимуществами над командой ERROR, поскольку в любом случае обработчик ошибок должен быть способен захватить его, или будет вызываться обработчик ошибок VFP. Фактически, если что-либо, кроме структуры TRY, захватывает THROW, то это может до некоторой степени сбить с толку относительно происхождения реальной проблемы, поскольку только структура TRY может захватывать «переброшенный» объект Exception. В случае метода Error или процедуры ON ERROR, полученные параметры и результаты AERROR() будут связаны с оператором THROW и необрабатываемой исключительной ошибкой, а не с той причиной, по которой использовалась THROW. Некоторые из свойств объекта Exception помещены в столбцы 2 и 3 массива, заполненного AERROR(), так что обработчик ошибок может анализировать эти столбцы. Однако вряд ли можно считать этот способ правильным. Напротив, убедитесь, что команда THROW используется только тогда, когда она может быть захвачена структурой TRY.

Проблемы в раю

Одной из наибольших проблем в обработке ошибок в VFP является предотвращение ошибок, возникаю-

щих когда мы и так находимся в состоянии ошибки. Под «состоянием ошибки» я понимаю состояние, при котором возникшая ошибка уже захвачена методом Error объекта или обработчиком ON ERROR, но обработчик еще не вызвал RETURN или RETRY. Если что-то пойдет не так, когда ваше приложение находится в состоянии ошибки, то не существует способа подстраховать вас, напротив, пользователь получит диалоговое окно сообщения об ошибке VFP с кнопками Cancel и Ignore. Это означает, что весь ваш механизм обработки ошибок в целом должен быть наиболее свободной от ошибок частью вашего приложения, а еще вам нужно проверять вещи, которые могут оказаться не ошибками, а проблемами среды.

Например, представьте себе, что ваш обработчик ошибок заносит ошибку в таблицу, называемую ERRORLOG.DBF. Что же случится, если этого файла не существует? Вы должны проверить его наличие с помощью FILE() и, если он не существует, создать его. Что будет, если что-либо еще открыло его монопольно? Вы можете избежать этого, никогда не открывая его монопольно, но, чтобы быть абсолютно уверенным, можно предварительно использовать FOPEN(), чтобы посмотреть, можно ли открыть его, поскольку FOPEN() возвращает код результата операции, а не ошибку. Что, если ERRORLOG существует и вы можете открыть его, используя FOPEN(), но этот файл поврежден? К сожалению, не существует простого способа проверить это.

Видите проблему? Ваш обработчик ошибок настолько усложняется, проверяя всё, что может идти не так во время обработки ошибки, что сам становится потенциальным источником ошибок!

В предыдущих версиях VFP не существовало решения этой проблемы. Вы просто писали некоторый рациональный код, проверяли его, насколько это представлялось возможным, и затем надеялись на лучшее. К счастью, в VFP 8 у нас есть решение: включение вашего обработчика ошибок в структуру TRY. Поскольку любые ошибки, случающиеся в блоке TRY, захватываются блоками CATCH, теперь у нас есть «страховка» для нашего обрабатывающего ошибки кода. Вот простой пример:

```
on error do ErrHandler with error(), program(), lineno()
use MyBadTableName && такой таблицы нет
on error
procedure ErrHandler(tnError, tcMethod, tnLine)
* Заносит ошибку в таблицу ErrorLog.
try
  use ErrorLog
  insert into ErrorLog values (tnError, tcMethod, tnLine)
  use
* Игнорирует любые проблемы в нашем обработчике.
catch
endtry
return
```

Если таблица ErrorLog не существует, не может быть открыта, повреждена или не может быть использована, то будет выполняться блок CATCH. В данном случае в нем нет кода, поэтому ошибка в ошибке игнорируется.

Стратегия обработки ошибок

Давайте соберем в кучу эту информацию и проанализируем стратегию обработки ошибок в целом. Вот подход, который я теперь использую:

- Используйте три уровня обработки ошибок. Структуры TRY для обработки локальных ошибок, методы Error для инкапсулированной обработки ошибок уровня объектов и команда ON ERROR для глобальной обработки ошибок. Первые два уровня обрабатывают все ожидаемые ошибки, а непредвиденные пересылают на следующий уровень.
- Включите ваши обработчики ошибок в структуру TRY, чтобы предотвратить появление диалогового окна ошибки VFP, если что-либо идет не так в вашем коде обработки ошибок.
- Используйте шаблон проектирования Chain of Responsibility для вашего механизма обработки ошибок. Для получения подробной информации о нем посмотрите мою статью «Еще раз об обработке ошибок» в июльском номере за 1998 год или официальное описание на www.stonefield.com.
- Не включайте в структуру TRY все ваше приложение целиком. Если вы сделаете это, у вас не будет способа остановить это приложение в случае, если будет иметь место какая-либо ошибка.
- Не используйте THROW до тех пор, пока не убедитесь, что структура TRY захватывает ее.

Заключение

Возможности структурированной обработки ошибок, добавленные в VFP 8, обеспечивают нас тремя уровнями обработки ошибок: локальным, на уровне объектов и глобальным. Структурированная обработка ошибок позволяет уменьшить объем кода, связанный с распространением информации об ошибках, упрощает ваши приложения, делая их более «читабельными» и более управляемыми. Кроме того, появилась возможность разрешать некоторые сложные проблемы (такие, как непреднамеренный захват одним объектом ошибки, возникшей в другом). Я советую вам провести некоторое время, «играя» со структурированной обработкой ошибок, и считаю, что она становится сейчас ключевой частью вашей стратегии обработки ошибок в целом.

❑❑❑

Неувязочка? Теперь уже нет

Майк Хелланд (Mike Helland)

Способность инициализировать и реагировать на пользовательские события, которая появилась в VFP 8.0, является эволюционным развитием объектной модели Fox. В этой статье Майк Хелланд объясняет, как работает собственное связывание событий (event binding) и почему вам захочется использовать его.

RaiseEvent() и BindEvent() — две мои любимые функции в VFP 8.0. Эта пара позволяет разработчикам Fox расширять имеющуюся модель событий и создавать гораздо больше инкапсулированных классов с меньшей зависимостью от наследования. Давайте обсудим следующую программу, описывающую сотовый телефон и автомобилиста, как компоненты некоторой системы:

```
* 1. Создаем компоненты моей системы
 Local phone, driver
 phone = CreateObject("Phone")
 driver = CreateObject("Driver")
* 2. Связываем их вместе
 BindEvent(phone, "Ring", driver, "AnswerPhone")
* 3. И приводим систему в движение
 phone.IncomingCall()
 Define Class Phone As Custom
 Function IncomingCall()
 * Описываем специальную логику для получения звонка
 * Пусть некто знает, что поступил звонок
 RaiseEvent(this, "Ring")
 Return
 * Функция, необходимая для события Ring Event
 Function Ring
 Return
 EndDefine
 Define Class Driver As Custom
 Function AnswerPhone()
 * Здесь должна следовать логика для ответа по телефону
 MessageBox("Hello?", 32)
 Return
 EndDefine
```

Этот пример вначале инициализирует два класса — phone и driver. В конце метода IncomingCall() класса Phone находится следующий код: RaiseEvent(this, "Ring"). Когда он исполняется, объект инициирует событие. Событие в VPF 8.0 — это сигнал, информирующий приложение о том, что произошло нечто, заслуживающее внимания. Определяемые пользователем события могут быть чем угодно и могут быть названы любым допустимым именем метода. Вследствие этого данному классу требуется метод Ring(), даже если он пуст. Второй класс, driver, даже проще. Единственным методом этого класса является AnswerPhone(), просто отображающий MessageBox().

Интересным в этой программе является не то, как работают ее компоненты, а то, как она заставляет их работать вместе. После того, как созданы объекты, необходимые для завершения задачи, на втором шаге программа использует BindEvent для «связи» события Ring() с методом AnswerPhone() класса driver. Проще говоря, программа утверждает: если телефон звонит, то водитель ответит.

На последнем шаге программа запускает IncomingCall(). Когда это происходит, класс phone порождает событие Ring (). Благодаря функции BindEvent(), VPF знает о событии «телефонный звонок». Далее должен автоматически выполниться метод AnswerPhone() класса driver, а потом приходит черед MessageBox().

(*Примечание относительно терминологии*: обычно функция Driver.AnswerPhone() называется «делегатом», поскольку задача ответа на звонок телефона делегируется этой функции. Сам класс driver называют «обработчиком событий» (Event Handler)).

Интересно, но ...

Заметили ли вы, что наши классы не ссылаются на другие классы или объекты и, что еще важнее, друг на друга? Класс phone ничего не знает о классе driver и наоборот. Несмотря на это, все компоненты прекрасно работают вместе в нашей системе.

Раньше, до появления VFP 8.0, можно было просто выполнить один за другим оба метода, Incoming() и AnswerPhone(), и наблюдать то же самое поведение, что и у исходной системы. Однако это будет означать, что программа должна уметь точно определять, когда выполняется метод AnswerPhone(). Это становится проблемой, когда IncomingCall() исполняется откуда-либо, кроме вершины стека вызовов. Нет гарантии, что у нас всегда будет ссылка на объект driver только потому, что существует ссылка на объект phone. Предположим, что третий шаг в коде нашей исходной системы был изменен следующим образом:

```
* 3. И приводим систему в движение
 callPhone(phone)
 Procedure callPhone(phone)
 phone.IncomingCall()
 EndProc
```

В процедуре callPhone() нет ссылки на объект driver. И не должно быть, поскольку только процедура callPhone() заинтересована в прямом общении с

объектом phone. Поэтому в этом сценарии невозможно вызвать DriverAnswerPhone() немедленно по завершении метода IncomingCall(). К счастью, благодаря реализации BindEvent(), эта система продолжает прекрасно работать.

Гораздо логичнее для этой программы описать, как объект driver будет реагировать на события объекта phone. И действительно, в реальном мире обычаи общества диктуют, как реагировать на звонок телефона.

Делегирование также означает гибкость. Представьте, что вам необходимо добавить следующий класс в систему, с которой мы работали:

```
Define Class Speaker As Custom
 Function PlaySound()
 ?? Chr(7)
 Return
 EndDefine
```

Используя связывание событий, системе требуется лишь инициализировать класс и «связать его»:

```
speaker = CreateObject("Speaker")
BindEvent(phone, "Ring", speaker, "PlaySound")
```

Добавление двух новых строчек, приведенных выше, это все, что необходимо для добавления функциональных возможностей класса speaker в нашу программу. Если бы реализация предполагала последовательность вызова методов, все равно добавление этой функциональной возможности не могло бы быть столь простым! Даже реализации, основанные на наследовании, не могут соединить вместе функциональные возможности настолько легко.

Причины использования Event Binding становятся более очевидными, как только сложность нашей системы возрастает. Давайте изменим нашу систему на что-нибудь более соответствующее реальному приложению. Пусть объект phone станет таймером, который ищет командный файл на жестком диске. Когда нужная команда найдена, генерируется событие, и объект, который может выполнить эту команду (объект driver), обрабатывает это событие. Альтернативные реализации объекта phone могут легко опрашивать Web Service, сканировать FTP-сайты или принимать входные данные с Web-страницы ASP.NET.

Несвязанные события

RaiseEvent() and BindEvent() — это только половина новых функций, имеющих отношение к связыванию событий в VFP 8.0. Две другие функции, UnBindEvent() и AEvents(), используются для явного удаления одной конкретной связки события и объекта или всех связок для конкретного объекта. Но это редко востребовано, поскольку после уничтожения объекта об очистке связей между событиями позаботится Fox. Оперативная справка очень подробно описывает поведение всех четырех функций.

Передача параметров

Для лучшего описания контекста события, RaiseEvent() поддерживает передачу параметров делегату. Эти параметры принимаются функцией обработчика как обычные параметры функции, посредством LParameters или через сигнатуру метода. Следующий пример кода создает таблицу со всеми днями недели. Затем таблица просматривается, и отображается день недели. Особенность заключается в том, что код, знающий как отображать сообщение, и код, знающий как просматривать таблицу, полностью разделены на два разных класса:

```
* Создаем курсор с семью записями и отображаем их
 Local i
 Create Cursor days (name c(10))
 For i = 1 To 7
   Insert Into days (name) VALUES (Cdow(Date() + i))
 Next i
 Local scanner, processor
 scanner = CreateObject("Scanner")
 processor = CreateObject("Processor")
 BindEvent(scanner, "ProcessRecord", processor, ;
   "ShowRecord")
 scanner.Scan()
 Define Class Processor As Custom
 * Показывает имя пересланной записи
 Function ShowRecord(message As String)
 MessageBox(message)
 Return
 EndDefine
 Define Class Scanner As Custom
 Function ProcessRecord(oRecord As Object)
 Return
 * Просматривает текущий выбранный псевдоним
 Function Scan(alias As String) As Boolean
 Local oRecord
 If Used()
   Scan
     * Извлекает событие для каждой записи и
     * пересылает эту запись делегату
     RaiseEvent(this, "ProcessRecord", name)
   EndScan
 EndIf
 Return .T.
 EndDefine
```

Пример представляет идею передачи сообщения от объекта scanner к объекту processor. Это сделано с помощью передачи имени поля как третьего параметра в RaiseEvent(). Затем этот параметр работает, как если бы он был обычным параметром, переданным функции ShowRecord(). И снова, это может выглядеть как слишком «раздутый» кусок кода, нечто подобное могло бы выглядеть и так:

```
Scan
   MessageBox(name)
EndScan
```

Но эту модель делает привлекательной ее гибкость. Самым большим ее преимуществом является то, что при необходимости исполняемый код можно быстро добавить или убрать из системы. Но самой интри-

гующей возможностью является класс Scanner и, что еще важнее, многочисленные его реализации. У нас есть сканер для курсоров, но мы также можем разработать сканер для XML-файлов, коллекций, разделенных запятыми строк, массивов, HTML screen scrap, и этот список можно продолжить. Это означает, что независимо от того, с какими данными вы работаете и как их храните, можно пройти по ним при помощи следующего кода:

```
Local scanner, processor
scanner = CreateObject("Scanner")
processor = CreateObject("Processor")
BindEvent(scanner, "ProcessRecord", processor, ;
  "ShowRecord")
scanner.Scan()
```

Хотя обычно невозможно вернуть значение от делегата, все же для этого делегата существует возможность обратной связи с событием при помощи настройки свойств объектных ссылок. Вернемся к примеру с объектом Scanner. Вместо передачи «имени» поля обработчику события, ему будет передано объектное представление этого объекта. Мы также воспользуемся новым свойством функции AddProperty() в VFP 8.0, чтобы добавить свойство Failed к параметрам объекта.

```
Scatter Name oRecord
AddProperty(oRecord, "Failed", .F.)
RaiseEvent(this, "ProcessRecord", oRecord)
If oRecord.Failed
  Return .F.
EndIf
```

В этой версии кода класс будет сканировать до тех пор, пока обработчик событий не установит для свойства Failed значение True. Если мы изменим наш класс Processor таким образом, чтобы выполнять присвоение, когда, к примеру, отображаемое сообщение было “Thursday”, мы можем эффективно сократить сканирование. Ниже приведен код, который мог бы сделать эту работу:

```
* Показывает имя пересланной записи
Function ShowRecord(oRecord As Object)
If oRecord.Name = "Thursday"
  oRecord.Failed = .T.
EndIf
MessageBox(oRecord.Name)
Return
```

Заключение

На прощание несколько примеров. Объект form может программно создать панель инструментов с кнопками и связывать событие Click каждой кнопки с методом объекта form, исключая необходимость создания специального класса toolbar. Объект слоя обработки данных, формирующий событие при изменении значения поля, может быть обработан следующим образом: 1) делегат среднего уровня убеждается в достоверности данных; 2) делегат обновляет пользовательский интерфейс, даже если эти объекты размещены на разных серверах; 3) делегат проверяет модификации, сделанные пользователем; 4) делегат реплицирует модификации на другой сервер.

Высокоуровневые службы, такие как MS Agent, могут без усилий реагировать на события в приложении. Также эта возможность может быть удалена без воздействия на другие компоненты. Я чувствую себя свободным от строгих иерархий наследственности и достигаю новых уровней инкапсуляции в коде.

❑❑❑

FoxTalk *русское издание*

Печатается ежемесячно

Учредитель и издатель:

ООО Эдэль.

Страничка в Интернете: http://newsletter.narod.ru или http://msnhomepages.talkcity.redmondave/dartemov/foxtalk.htm

(095) 325-5278
E-mail: foxtalk@online.ru
115304 Москва, а/я 208

Главный редактор: Д. Артемов
E-mail: dartemov@hotmail.com

Журнал зарегистрирован комитетом Российской Федерации по печати.

Регистрационное свидетельство № 015520 от 17.12.1996

Подписано в печать 10/08/03. Формат 60x90 1/8. Тираж 300 экз.