7

Создание надежного кода

Надежный код можно сравнить с Суперменом и бронежилетом. Супермен неуязвим (если рядом нет криптонита). Бронежилет также делает владельца неуязвимым, пока кто-то не воспользуется бронебойными пулями (или, что интересно, композитным луком). Иногда неуязвимость рассматривается как что-то нереальное и невозможное, а иногда неуязвимость вполне возможна, но с определенными ограничениями.

Существующая технология не позволяет создавать алгоритмы, которые математически доказывают эффективность приложения, поэтому невозможно доказать, что оно совершенно лишено ошибок или неуязвимо. Все считают, что Супермен сделан из стали и непобедим, а представители силовых структур просто носят бронежилеты. Разработчики также применяют средства защиты при создании кода приложений. В этой главе рассматриваются методики, обеспечивающие жизнеспособность приложения и позволяющие защитить его от аварийного завершения работы, удаления файлов или более нежелательных вариантов поведения. Если что-то пойдет не так, можно будет воспользоваться описанными здесь приемами диагностики и решения проблемы. Кроме этого, описанные подходы являются переносимыми и могут применяться при создании различных приложений.

Обсуждение основано на проверенной десятилетиями стратегии создания защищенного кода с помощью инструментов, предоставленных интерпретатором VBA. Здесь рассматриваются способы обнаружения неоправдавшихся предположений и методика контроля за порядком передачи управления в пределах кода, что позволяет оставлять “хлебные крошки” и проверять каждый путь, ветвление и цикл кода. В результате применения этих методик остается меньше потенциальных проблем. Если проблемы все же возникают, разработчик может воспользоваться доступной информацией для их быстрой диагностики и решения.

Использование метода Debug.Print

Объект Debug предоставляет несколько методов. Краеугольным камнем фундамента инструментов отладки и тестирования является метод Debug.Print. Еще один метод, Debug.Assert, будет рассматриваться в следующем разделе.

Debug.Print — очень простой метод. Он принимает строку с сообщением и отправляет сообщение в окно Immediate (Проверка). Кроме этого, метод Debug.Print выполняет свою задачу только при запуске кода в режиме отладки (в редакторе VBE), что делает его идеальным инструментом для отслеживания реального текущего состояния кода, а не предполагаемого.

Как в свое время сказал Рональд Рейган: "Доверяй, но проверяй". Мы, как разумные программисты, верим, что код будет выполняться в том порядке, который *имел в виду* программист, но на самом деле код всегда выполняется в том порядке, который *был записан* программистом. Метод Debug.Print позволяет точно узнать, как выполняется код.

Создадим инструментарий отладки, воспользовавшись в качестве основы методом Debug.Print. Вызовы Debug.Print можно вставлять в любой метод или свойство, например:

```
Debug.Print "Шляпу можно не снимать"
```

Более полезным использованием метода является вывод имен объекта и метода, а также текста с полезной информацией о состоянии до и после выполнения интересующего оператора. Вот метод, который определен в условном листе Sheet1 и использует вызов Debug.Print:

```
Public Function CalculateFuelConsumed(ByVal Duration As Double, _
  ByVal GallonsPerHour As Double) As Double

  Debug.Print "Вошли в Sheet1.CalculateFuelConsumped"

   CalculateFuelConsumed = Duration * GallonsPerHour

  Debug.Print "Выходим из Sheet1.CalculateFuelConsumped, результат = " & _
    CalculateFuelConsumed & " галлонов"

End Function
```

Предыдущий метод вычисляет объем топлива, которое потребляется за время работы продукта, а также объем топлива, потребляемого за один час. Вызов Debug.Print используется для создания вывода в окне Immediate (Проверка), показанного на рис. 7.1.

Рис. 7.1. Результат работы метода Debug.Print в окне Проверка

Помните, что объект Debug работает только в редакторе VBE. В результате отладочный код можно не удалять, так как пользователи все равно не узнают о его существовании. Преимуществом такого подхода является наличие доступности кода при возникновении любой проблемы. При этом разработчики избавляются от необходимости добавлять и удалять код между циклами отладки и сопровождения.

Метод Debug.Print еще будет рассмотрен в разделе "Отслеживание ошибок".

Использование метода Debug.Assert

Еще одним методом объекта `Debug` является метод `Assert`. Этот метод выступает в роли наемного стрелка на стороне программиста и предназначен для остановки выполнения кода в среде разработки, если поставленное методом `Assert` условие не выполняется.

При создании кода программисты строят явные и неявные предположения о состоянии приложения на определенных этапах. Примером явного предположения является проверка существования файла перед его открытием для чтения. Неявным предположением является неповрежденность дискового кластера, в котором хранится необходимый файл. В любом случае, программист знает о явных и неявных предположениях только при создании кода и появлении "наглой морды" неявного предположения. Если предполагается, что что-то плохое может произойти, лучше сразу нанять стрелка, который будет патрулировать окрестности. Метод `Debug.Assert` является лучшим стрелком, нанятым программистом на Excel VBA.

При создании каждого метода желательно добавить условный код, проверяющий соответствие некоторых независимых параметров преполагаемым условиям. Например, деление на 0 возникать не должно. Поэтому при выполнении деления желательно убедиться, что делитель не равен 0. Но так как деление на 0 никогда не должно происходить, желательно проинструктировать стрелка о существовании этого предположения. Помните, что объект `Debug` является инструментом программиста. Метод `Debug.Assert` завершит работу приложения, если условие не будет выполняться, но объект создается только при запуске кода в редакторе VBE в процессе отладки. По этой причине объект `Debug` никогда не станет полноценной заменой нормальным проверкам. Он просто, с точки зрения программиста, дополняет проверки. В отличие от вызова метода `Debug.Assert` условие `If` не остановит работу приложения, если условие не выполняется. Следовательно, условный оператор `If` может защитить код от взрыва, а объект `Debug` сообщает о возможности взрыва во время отладки. В комбинации эти инструменты позволят получить более мощные средства отладки, как показано в следующем фрагменте кода:

```
Public Sub DivideByZero(ByVal Numerator As Double, _
  ByVal Denominator As Double)

  Dim result As Double

  Debug.Assert Denominator <> 0
  If (Denominator <> 0) Then
    result = Numerator / Denominator
  Else
    ' сделать что-то другое
  End If
End Sub
```

Если предположение оказывается неверным в редакторе VBE, то выполнение остановится на строке `Debug.Assert Denominator <> 0`. Программист сразу поймет, что потребитель (программист, который воспользовался методом `DivideByZero`) нарушил необходимое условие и присвоил параметру `Denominator` значение 0 (рис. 7.2).

В следующем разделе будет показано, как использовать методы `Debug.Print` и `Debug.Assert` для создания инструментария многоразового использования, чтобы получить защищенный код. Если время на чтение ограничено, можно пропустить разделы "Краткая история отладки на ПК" и сразу переходить к разделу "Создание многоразовых инструментов на основе объекта Debug".

Краткая история отладки на ПК

История очень важна, так как позволяет охватить проблему в более широкой перспективе. Интересно, но история микрокомпьютеров насчитывает не более 20 лет. (Первый компьютер IBM PC поступил в продажу в августе 1981 года.) Таким образом, работавшие в 1981 году программисты являются живыми свидетелями сжатой истории микрокомпьютеров.

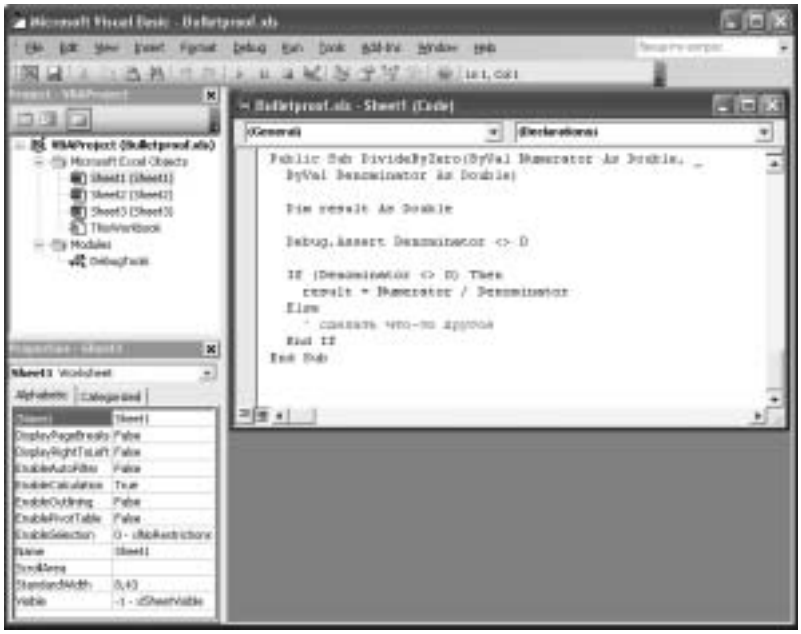

Рис. 7.2. Остановка приложения при неподтвердившемся предположении

Примерно в то время Тим Патерсон (Tim Paterson) продал собственную дисковую операционную систему Биллу Гейтсу за $25000. Впоследствии она стала называться MS-DOS. Персональный компьютер работает под управлением комбинации из дисковой операционной системы и программы BIOS (система базового ввода-вывода). Эти два компонента предоставляют базовые строительные блоки, позволяющие компьютеру работать, а программистам — создавать программы.

Программа BIOS обеспечивает работу прерываний, которые представляют собой глобальные функции, загружаемые вместе с базовой подсистемой ввода-вывода во время включения компьютера. Эти функции существуют до сих пор под покровом нескольких слоев сложного кода Windows. Прерывания имеют номера 0, 1, 2, 3 и т.д. Можно предположить, что прерывания с меньшими номерами появились раньше, а прерывания с большими номерами появились позже, вместе с появлением дополнительных возможностей. Например, прерывание DOS имеет номер 0x21 (обычно для нумерации прерываний используются шестнадцатеричные числа; данное прерывание имеет десятичный номер 33). Учитывая предположение о первоочередном появлении прерываний с меньшими номерами, не удивительно, что прерывание с номером

0 соответствует ошибке деления на ноль. Деление является важной операцией для программистов, а деление на 0, похоже, было значительной проблемой, если оно обрабатывается одной из самых базовых системных служб.

Назначение прерываний 1 и 2 мало кого интересует. Прерывание 3 оказывается более важным с точки зрения текущего обсуждения, так как это прерывание отладки. Оно останавливает выполнение программы. Скорее всего, именно это прерывание лежит в основе оператора `Stop` в языке VBA и точек останова в редакторе VBE. (В более ранних версиях отладочных инструментов прерывание 3 называлось "soft-ice"; то есть, вызов этого прерывания приводил к "замораживанию" программы.)

Все эти базовые возможности до сих пор существуют и используются компьютером для выполнения поставленных ранее задач, хотя над этими прерываниями и создан большой объем кода. Убедиться в существовании этих возможностей можно с помощью команды `debug.exe` в приглашении командной строки. Для этого можно написать простой машинный код. (Будьте осторожны, так как базовые возможности обладают большой мощностью.)

Руководствуясь рис. 7.3, введите следующие отладочные инструкции и код ассемблера. В результате будет создана программа на языке ассемблера. (Этот пример является хорошим напоминанием о том, насколько более удобным инструментом является язык VBA.)

Рис. 7.3. Ввод программы в отладчике debug.exe

```
debug
nhello.com
a100
jmp 110
db "Hello, World!$"
mov dx,102
mov ah,9
int 3
int 21
int 20

rcx
1a
w
g
q
```

Ниже по порядку приводятся инструкции и код.

- Запустите программу `Debug`. Для этого в приглашении командной строки введите команду **`debug.exe`**. Это очень простой отладчик и редактор, который иногда может оказаться очень мощным.
- Введите **`nhello.com`**. Эта инструкция сообщает редактору `debug.exe` имя файла, в который необходимо записывать вывод.
- Введите **`a100`**. Это инструкция языка ассемблера. Она сообщает редактору, что начинается ввод кода.
- Введите **`jmp 110`**. Эта команда является аналогом нашего хорошего друга — команды `GOTO` на языке ассемблера.
- Команда **`db "Hello, Word!$"`** используется для объявления строковой переменной.
- Команда **`mov dx,102`** заносит значение 102 (шестнадцатеричное значение) в регистр DX центрального процессора. Регистры являются аналогами переменных на самом нижнем уровне. Вообще это переключатели внутри микропроцессора, которые используются миллион раз в секунду. На этом этапе в регистр DX заносится адрес текстовой строки.
- Команда **`mov ah,9`** в данном контексте является функцией. В целом эти команды используются для подготовки вызова функции, аргументом которой является строка `"Hello, Word!"` (символ `$` является символом окончания строки), а 9 является номером функции.
- Команда **`int 3`** выполяет функцию точки останова. Код будет выполняться до команды **`int 3`**, после чего выполнение будет приостановлено, и выдано текущее состояние процессора. (Текущее состояние показано на рис. 7.3.)
- Команда **`int 21`** является прерыванием DOS. Прерывание 21 и функция 9 используются для вывода строк.
- Вызов прерывания **`int 20`** приводит к завершению работы программы. Он сообщает процессору о необходимости передать управление операционной системе.
- Пустая строка введена намеренно. Ввод пустой строки возвращает редактор из режима программирования в режим управления.
- Команда **`rcx`** является инструкцией отладчика, позволяющей вывести и отредактировать значение регистра CX, который используется, чтобы сообщить отладчику количество записываемых байт.
- **`1a`** является шестнадцатеричным числом (в десятичной форме 26). Отладчик должен записать 26 байт ассемблерного кода.
- Команда **`w`** является инструкцией на запись.
- Команда **`g`** является аналогом клавиши <F5> в среде разработки VBA. Эта команда приводит к выполнению программы в режиме отладчика.
- Команда **`q`** приводит к выходу из программы. Введите **`q`** и нажмите клавишу <Enter>. Работа утилиты `debug.exe` завершится.

После завершения работы утилиты будет создана программа `hello.com`. Файл будет храниться во временном каталоге или в каталоге, который был указан в начале работы программы. Введите **`Hello`** в приглашении командной строки. Обратите внимание на выведенный текст `Hello World!`. После этого программа возвращает управление интерпретатору командной строки. Кроме этого, обратите внимание на то, что точка останова была проигнорирована. Не кажется ли знакомым это поведение? Да, верно. Оно характерно для объекта `Debug`, который создается только при отладке в редакторе VBE. За пределами редактора VBE вызовы методов объекта `Debug` игнорируются. Можно считать, что принципы отладки были показаны до самого нижнего, аппаратного, уровня. (Ниже только физические механизмы в кристалле процессора, но рассматривать их будет лишним.) Теперь понятно, почему деление на ноль и точки останова были так важны для ранних программистов. Кроме этого, ясно, что эти возможности все еще используются в компьютере, хотя они и скрыты за более удобными средами вроде VBE.

Создание многоразовых инструментов на основе объекта Debug

Одной из любимых авторами книг по программированию является *No Bugs!* Дейва Тилена (Dave Thielen), которая выпущена издательством Addison-Wesley. Книга для программистов на C была написана после успешного выхода операционной системы MS-DOS 5.0. Хотя C является языком более низкого уровня по сравнению с VBA, большинство приемов отладки основаны на тех же принципах, поэтому данные приемы можно использовать в VBA вместе с объектом `Debug`. Важность этих подходов связана с тем, что они работают всегда и их работоспособность проверена временем.

В этом разделе рассматривается три мощных управляемых инструмента, которые позволяют систематически исправлять ошибки и быть уверенными, что обнаруженные ошибки исправлены. Если что-то пойдет не так, эти инструменты обнаружат и исправят проблему немедленно. Они называются `Trace`, `Trap` и `Assert`. Как компания Microsoft создала объект `Debug` и оператор `Stop` поверх базовых возможностей BIOS (как показано в разделе “Краткая история отладки на ПК”), так и мы создадим нашу реализацию поверх объекта `Debug` и оператора `Stop`.

Определение последовательности выполнения

Определение последовательности обеспечивается размещением строк кода, сообщающих информацию о маршруте и состоянии кода в определенный момент. Эта информация может оказаться настолько полезной в процессе отладки, что многие инструменты разработчика автоматически отслеживают выполнение кода, создавая стек вызовов (определяя последовательность вызова методов) и показывая последовательность выполнения. Для определения последовательности выполнения в VBA может использоваться метод `Debug.Print`, но автоматическое отслеживание порядка вызова методов при этом не поддерживается. Это придется делать вручную. Можно создать собственную версию подпрограммы `Trace` и описать интересующую информацию, получая одинаковый по форме вывод при каждом использовании этого инструмента. Вот метод `Trace`, реализованный в экспортируемом модуле `DebugTools.vb`.

```
Option Explicit
#Const Tracing = True
#Const UseEventLog = False

Private Sub DebugPrint(ByVal Source As String, _
  ByVal Message As String)

#If UseEventLog Then

#Else
  Debug.Print "Источник: " & Source & " Сообщение: " & Message
#End If

End Sub
Public Sub Trace(ByVal Source As String, _
  ByVal Message As String)
#If Tracing Then
  Call DebugPrint(Source, Message)
#End If

End Sub
```

В предыдущем фрагменте кода была определена константа компилятора: `Tracing`. Метод `DebugPrint` является универсальным методом, который принимает в качестве параметров две строки. Параметры называются `Source` и `Message`. Обратите внимание на закомментированную часть кода, в которой константа `UserEventLog` управляет вызовом метода `Debug.Print`. (К условию, зависящему от константы `UserEventLog`, возвратимся в разделе "Запись в журнал событий" далее в этой главе.) Последний метод, `Trace`, принимает те же два аргумента: `Source` и `Message`. Метод `Trace` проверяет значение константы отладчика. Если константа равна `True`, вызывается процедура `DebugPrint`.

Можно спросить: зачем использовать константы компилятора и заворачивать вызов метода `Debug.Print`, если объект `Debug` автоматически отключается при запуске кода за пределами VBE? Хороший вопрос.

Программирование очень похоже на выращивание лука изнутри. Программирование начинается с ядра, например базовой подсистемы ввода-вывода компьютера. Постепенно слои накладываются поверх существующих уровней, медленно, но уверенно добавляя сложности. (Сложность может оказаться субъективным параметром.) Эти уровни должны быть небольшими, простыми для тестирования и предоставлять необходимый уровень функциональности, которая не должна быть сложной или большой по объему. Причиной постепенного добавления функциональности является попытка снизить сложность реализации изменений и минимизация вероятности появления дополнительных дефектов. Кроме этого, простота позволяет легко оценить полезность вносимых изменений. Другими словами, большие объемы кода (большие монолитные реализации) являются причиной головных болей в будущем.

Однозначное поведение, построенное поверх более простого однозначного поведения, как ламинированное дерево, позволяет построить мощные и полезные реализации, которые сохраняют простоту на каждом уровне. Именно так выглядит хороший, полезный, поддающийся отладке и защищенный код. К сожалению, эта особенность программирования не рассматривается в институтах и университетах. (Просим извинения у тех, чье учебное заведение было исключением.) Это связано с тем, что данная отрасль все еще молода и не все еще приняли правильное решение по этому вопросу.

(Стоит обратить внимание, что большинство успешных специалистов в данной области все-таки используют ту или иную аналогию со слоями лука.) Создание многоуровневого, однозначного кода требует определенной предварительной практики, но именно для этого написана данная книга.

Помня о необходимости создания сложных систем в виде нескольких простых уровней следует создать простой метод `Trace`. Метод принимает два строковых параметра. Кроме этого, метод предоставляет отдельную возможность включения и отключения отладки. На основе поведения редактора VBE определение отслеживания вызовов было расширено до использования другого хранилища. Вместо ведения журнала в окне Immediate (Проверка) был добавлен код для записи журнала в более постоянное хранилище, в журнал событий. После этого необходимо научиться использовать новое поведение метода `Trace`.

Программист должен выбрать отслеживаемые параметры приложения. Можно отслеживать все параметры, но это слишком утомительно. Вместо этого необходимо добавить вызовы `Trace` в тех местах, которые особенно важны для реализации, а также в местах, где возникают ошибки. После добавления вызова `Trace` его можно оставить даже после исправления всех ошибок. При внесении изменений в код могут появиться новые ошибки, но отладочный код уже будет присутствовать в ключевых местах. Одним из преимуществ вызова `Trace` является обозначение фрагментов кода, которые интересовали разработчиков в прошлом. То есть, вызов `Trace` выступает в качестве “хлебных крошек”, которые разработчики оставляют для того, чтобы вернуться обратно, если возникнет необходимость. Вот метод `CalculateFuelConsumed` из предыдущей главы. Вызовы метода `DebugPrint` заменены на вызовы `Trace`:

```
Public Function CalculateFuelConsumed(ByVal Duration As Double, _
  ByVal GallonsPerHour As Double) As Double

  Call Trace("Sheet1.CalculateFuelConsumped", _
    "Длительность=" & Duration & " Галлонов в час =" &
↳ GallonsPerHour)

  CalculateFuelConsumed = Duration * GallonsPerHour
  Call Trace("Sheet1.CalculateFuelConsumped", "Результат=" &
↳ CalculateFuelConsumed)

End Function
```

Конечным результатом являются одинаковые вызовы `Trace` и одинаковые правильно отформатированные сообщения. На рис. 7.4 показано окно Immediate (Проверка) после вызова метода `CalculateFuelConsumed`.

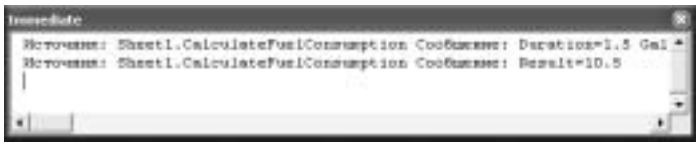

Рис. 7.4. Результат использования метода Trace

Получение маршрута выполнения кода

Следующий метод называется перехватом. Необходимо проверять все маршруты, по которым может выполняться код — условия `If` и `Else`, все методы и свойства, циклы `While`, тело которых выполняется и не выполняется. Если этого не сделать, то нельзя быть уверенным, что непроверенный маршрут не приведет к опасному поведению приложения.

Идиома `Trap` используется для отметки веток кода, чтобы обеспечить тестом каждый закоулок кода и получить ожидаемый и необходимый результат. Идиома `Trap` находится уровнем выше оператора `Stop` и, как и метод `Trace`, помещается в оболочку для получения дополнительной гибкости. Вот код реализации идиомы `Trap`, который можно добавить в модуль `DebugTools`. После этого будет показано, как создавать ловушки:

```
Option Explicit
#Const Tracing = True
#Const Debugging = True
#Const UseEventLog = False

Private Sub DebugPrint(ByVal Source As String, _
  ByVal Message As String)

#If UseEventLog Then

#Else
  Debug.Print "Источник: " & Source & " Сообщение: " & Message
#End If

End Sub
Public Sub Trap(ByVal Test As Boolean, _
  ByVal Source As String, _
  ByVal Message As String)

#If Debugging Then
  If (Test) Then
       Call DebugPrint(Source, Message)
       Stop
     End If

#End If
End Sub
```

В подпрограмме `Trap` используется константа `Debugging` и метод `DebugPrint`, уже применяемый ранее. Не будем еще раз описывать назначение этих элементов. Рассмотрим особенности метода `Trap`.

В методе `Trap` используются те же два аргумента, что и в методе `Trace`. Это `Source` и `Message`. Они нужны для идентификации сработавшей ловушки и для поиска соответствующего вызова метода `Trap`. При срабатывании ловушки источник срабатывания и сообщения заносятся в журнал, а работа приложения завершается (рис. 7.5).

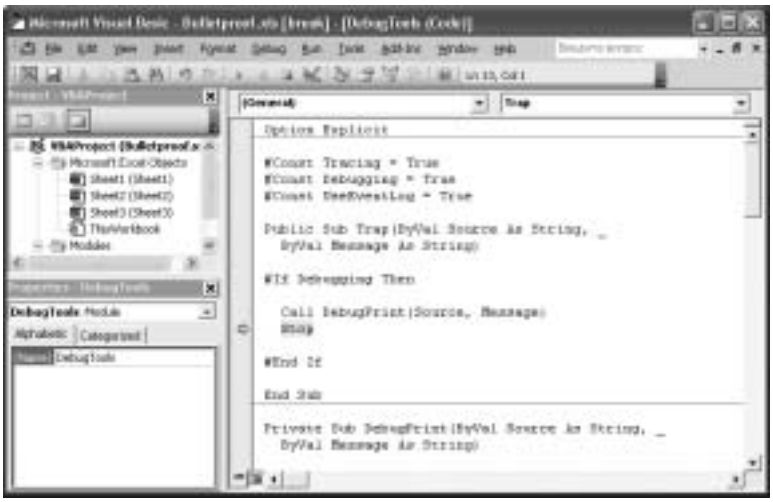

Рис. 7.5. Остановка приложения при срабатывании ловушки

После срабатывания ловушки значение параметра `Source` можно использовать для поиска и комментирования сработавшего вызова. Комментирование конкретного вызова метода `Trap` указывает, что данный маршрут выполнения кода был протестирован и можно переходить к тестированию других веток кода. При этом закомментированный вызов метода `Trap` остается внутри кода на случай, если тестирование придется повторить. Например, для проверки существования нового теста для метода `CalculateFuelConsumed` в этот метод можно добавить вызов метода `Trap`. Вот как будет выглядеть модифицированная версия метода `CalculateFuelConsumed`:

```
Public Function CalculateFuelConsumed(ByVal Duration As Double, _
  ByVal GallonsPerHour As Double) As Double

  Call Trap("Sheet1.CalculateFuelConsumed", "Tested")

Call Trace("Sheet1.CalculateFuelConsumed", _
    "Длительность=" & Duration & " Галлонов в час =" &
➥ GallonsPerHour)

  CalculateFuelConsumed = Duration * GallonsPerHour

  Call Trace("Sheet1.CalculateFuelConsumed", "Результат=" &
➥ CalculateFuelConsumed)

End Function
```

После создания теста для метода `CalculateFuelConsumed` оператор вызова метода `Trap` можно закомментировать.

Чистоплотные разработчики могут не согласиться со смешиванием отладочного кода и кода, выполняющего поставленную перед приложением задачу. В таком случае лучше принять стратегию разделения между реальным и отладочным кодом. Здесь будет использоваться стратегия создания закрытого теневого метода с префиксом `Do`. Реальный код добавляется в метод с префиксом `Do`, а в методе с обычным именем бу-

дет находиться отладочный код. В результате выполнение алгоритма будет происходить отдельно от функциональности бронежилета — отслеживания, перехвата и кода проверки предположений. Вот исправленный вариант кода:

```
Public Function CalculateFuelConsumed(ByVal Duration As Double, _
  ByVal GallonsPerHour As Double) As Double

  'Call Trap("Sheet1.CalculateFuelConsumed", "Tested")

  Call Trace("Sheet1.CalculateFuelConsumed", _
    "Длительность=" & Duration & " Галлонов в час =" &
↳ GallonsPerHour)

  CalculateFuelConsumed = DoCalculateFuelConsumed(Duration, _
                          FuelConsumedPerhour)

  Call Trace("Sheet1.CalculateFuelConsumed", "Результат =" &
↳ CalculateFuelConsumed)

End Function

Private Function DoCalculateFuelConsumed(ByVal Duration As Double, _
  ByVal GallonsPerHour As Double) As Double

  DoCalculateFuelConsumed = Duration * GallonsPerHour

End Function
```

Неискушенный наблюдатель может решить, что дополнительный код выглядит неуклюже. Так и есть. Но тут, как и в случае с бронежилетом, стоит обеспечить максимальную дополнительную защиту. В итоге подобная структура кода продемонстрирует профессиональный подход к решению задачи. После завершения создания отладочного инструментария `DebugTools` можно рассчитывать на формирование привычки. А сформированная привычка значительно упростит написание такого кода. Если вам повезло работать с программистами различного уровня, то вы можете создавать реализацию фактического поведения, а младший программист может заворачивать его в оболочку отладочного и тестового кода. Кроме этого, средства отладки могут быть интегрированы в код при появлении проблем.

Проверка инвариантных условий

Наконец пришло время рассмотреть метод проверки предположений. Так получилось, что нам нравятся полицейские, но если кого-то беспокоит аналогия с органами охраны правопорядка, помните, что программирование можно представить как инструктирование компьютера и определение границ контроля над выполнением. Без открытых и закрытых методов не было бы методов защиты данных. Без приемов отладки не было бы способов решения проблем. Поэтому программист, как полицейский на параде, старается предотвращать проблемы и следить за правильной работой кода.

Предположение является обученным, одетым в бронежилет и форму, полицейским офицером. При создании кода программист делает базовые предположения о состоянии и поведении кода. Поведение `Assert` позволяет обеспечить сохранение истинности предположений. Результат похож на утопию, где нет вождения в нетрезвом виде, нецензурной брани и хулиганских выходок.

Метод Assert стоит использовать в тех местах кода, где предположения всегда должны быть истинными. Для сохранения однородности заключите метод Assert в оболочку DebugTools. Это позволит определиться с ожиданиями и обнаружить места, где ожидания не оправдываются. Вот пример кода:

```
Public Sub Assert(ByVal Test As Boolean, _
  ByVal Source As String, ByVal Message As String)

#If Debugging Then
  If (Not Test) Then
       Call DebugPrint(Source, Message)
       Debug.Assert False
     End If
#End If

End Sub
```

Метод Assert использует константу Debugging, которая определена ранее в этой главе, и метод DebugPrint. Метод Assert принимает параметр типа Boolean и еще два параметра: Source и Message. Сначала необходимо убедиться, что приложение работает в режиме отладки. После этого проверить истинность или ложность предположений. Если предположение не оправдалось, в журнал записывается сообщение и вызывается базовый метод Assert, который приводит к остановке работы приложения. Теперь рассмотрим проверку предположений в методе CalculateFuelConsumed. В данном случае необходимо убедиться, что параметры GallonsPerHour и Duration имеют неотрицательные значения. (В метод Do также добавлены условные операторы.)

```
Public Function CalculateFuelConsumed(ByVal Duration As Double, _
  ByVal GallonsPerHour As Double) As Double

  'Call Trap("Sheet1.CalculateFuelConsumption", "Tested")

  Call Trace("Sheet1.CalculateFuelConsumed", _
    "Длительность =" & Duration & " Галлонов в час =" & _
 GallonsPerHour)

  Call Assert(Duration > 0, "Sheet1.CalculateFuelConsumed", _
"Duration > 0")
  Call Assert(GallonsPerHour > 0, "Sheet1.CalculateFuelConsumed", _
    "GallonsPerHour > 0")

  CalculateFuelConsumed = DoCalculateFuelConsumed(Duration, _
FuelConsumedPerhour)

  Call Trace("Sheet1.CalculateFuelConsumption", "Результат =" &
↳ CalculateFuelConsumed)

End Function
Private Function DoCalculateFuelConsumed(ByVal Duration As Double, _
  ByVal GallonsPerHour As Double) As Double

  If (Duration > 0 And GallonsPerHour > 0) Then
    DoCalculateFuelConsumed = Duration * GallonsPerHour
  Else
    ' Здесь выдается сообщение об ошибке
  End If

End Function
```

В модифицированном примере добавлены два вызова метода `Assert`. Первый вызов сравнивает значение параметра `Duration` с 0. Обратите внимание, что в методе `DoCalculateFuelConsumed` добавлена проверка времени выполнения. В результате метод стал самодокументированным, имя описывает назначение, а известные механизмы тестирования делают метод достаточно защищенным. Вызов метода `Trap` добавлен для напоминания о необходимости тестирования. Глобальный поиск метода `Trap` позволяет найти незакомментированные вызовы и обнаружить непротестированные методы. Здесь используется два оператора `Trace`, которые показывают, где и в каком порядке используется этот метод. Наконец, добавлены предположения о допустимости значений параметров.

Вот еще один совет перед тем, как переходить к следующей теме. Что делать, если после написания определенного фрагмента кода и добавления вызовов `Trap` и `Trace` оказывается, что ни одна из ловушек не срабатывает? Ответ предполагает сброс мертвого балласта. Другими словами, удалите ненужный код. Он просто занимает дорогое время разработчиков, которые вынуждены читать на самом деле не представляющий интереса код.

Обратите внимание, что последний вариант метода содержит два оператора ветвления и блок `Else`, содержащий только комментарий. Необходимо добавить вызов `Trap` в каждую ветку условного оператора, чтобы протестировать оба маршрута выполнения кода. Кроме этого, нужно рассмотреть создание сообщения об ошибке. (Сообщения об ошибках будут рассматриваться в следующем разделе.) Далее показана последняя версия кода в этом разделе, в которой продемонстрирован удобный способ добавления операторов `Trap` в каждую ветку условного оператора:

```
Public Function CalculateFuelConsumed(ByVal Duration As Double, _
  ByVal GallonsPerHour As Double) As Double

  'Call Trap("Sheet1.CalculateFuelConsumption", "Tested")

  Call Trace("Sheet1.CalculateFuelConsumed", _
    "Длительность =" & Duration & " Галлонов в час =" &
↳ GallonsPerHour)

  Call Assert(Duration > 0, "Sheet1.CalculateFuelConsumed", _
"Duration > 0")
  Call Assert(GallonsPerHour > 0, "Sheet1.CalculateFuelConsumed", _
    "GallonsPerHour > 0")

  If (Duration > 0 And GallonsPerHour > 0) Then
    Call Trap("Sheet1.CalculateFuelConsumed", _
      "Duration > 0 And GallonsPerHour > 0")
    CalculateFuelConsumed = DoCalculateFuelConsumed(Duration, _
GallonsPerHour)
  Else
    Call Trap("Sheet1.CalculateFuelConsumed", "Duration > 0 And
↳ GallonsPerHour > 0 is False")
    ' Здесь выдается сообщение об ошибке
  End If

    Call Trace("Sheet1.CalculateFuelConsumption", "Result=" &
↳ CalculateFuelConsumed)

End Function

Private Function DoCalculateFuelConsumed(ByVal Duration As Double, _
  ByVal GallonsPerHour As Double) As Double
```

```
DoCalculateFuelConsumed = Duration * GallonsPerHour

End Function
```

Обратите внимание, что в последнем варианте кода вместе с предположениями используются операторы `If`, а весь отладочный код расположен во внешнем методе (не имеющем префикса `Do`). В результате алгоритм оказывается очень ясным и полностью защищенным.

Кто-то может поинтересоваться, нужно ли писать этот код для каждого метода? Ответ: нет. После написания пары миллионов строк кода появилась уверенность в том, что однострочный метод настолько прост в тестировании и отладке, что добавлять такой объем отладочного кода к столь простому методу абсолютно излишне. На самом деле, стоит пытаться сохранить простоту кода на уровне метода `DoCalculateFuelConsumed`. Кроме избежания большого количества ошибок такая простота избавляет от необходимости писать комментарии или отладочный код. Начинающие разработчики еще не так уверены в своих силах, а попав в трудную ситуацию, начинают сомневаться в правильном выборе путей выхода из нее. В итоге сам разработчик принимает решение о достаточном объеме отладочного кода. Это субъективное решение, которое и делает хорошее программирование таким сложным.

Вывод сообщений об ошибках

Существует много хороших программистов, которые не согласны с нашими рекомендациями. Это хорошо. Именно это и делает жизнь интересной и позволяет создавать новые идеи. Есть одна область, в которой сотни и даже тысячи программистов не могут прийти к единому решению. Надеемся убедить читателей в нашей правоте и в ошибочности подхода других программистов.

Много лет назад семантически более слабые языки, например C, возвращали коды ошибок при любом некорректном поведении. Коды ошибок представляли собой произвольные целые числа (обычно отрицательные), имевшие смысл только в определенном контексте. В итоге был создан большой объем кода, возвращавшего определенное целое число, имевшее смысл в определенном контексте. Такой подход к программированию предполагал, что все является функцией, все реальные возвращаемые значения передавались по ссылке (`ByRef`), а смысл кода ошибки не передавался вместе с ошибкой. Знание о смысле кода хранилось отдельно. Таким образом, если функция возвращала значение –1 в качестве кода ошибки, потребитель кода должен был обращаться к другому источнику для получения смысла кода ошибки –1. Вот переработанная версия метода `DoCalculateFuelConsumed`, которая возвращает код ошибки. (Такой код писать не рекомендуется.)

```
Private Function CalculateFuelConsumed(ByVal Duration As Double, _
  ByVal GallonsPerHour As Double, ByRef Result As Double) As Integer

  If (Duration > 0 And GallonsPerHour > 0)Then
    Result = Duration * GallonsPerHour
    CalculateFuelConsumed = 0
  Else
    CalculateFuelConsumed = -1
  End If

End Function
```

В данной версии метода возвращаемое значение указывает на успешность завершения работы, а фактический результат работы алгоритма передается через параметр. Это усложняет непосредственное использование функции, так как необходимо объявить дополнительный аргумент для хранения и получения результата работы. Теперь для вызова метода `CalculateFuelConsumed` необходимо писать значительно больше кода.

```
Dim Result As Double
If( CalculateFuelConsumed(7, 1.5, Result) = 0) Then
' Да: все сработало и Result можно использовать
Else
  ' Жаль, опять неудача и содержимое Result не имеет смысла
End If
```

Это особенно пессимистический подход к программированию. Полезность возвращаемого значения функции практически исчезает и приходится программировать так, как будто код всегда находится на грани падения. Продолжив чтение главы о создании защищенного кода, вы поймете, что код будет отказывать значительно реже. Вернемся к использованию возвращаемого значения функции и создадим оптимистический код, предполагая, что отказ является редкой неприятностью, а не частым гостем. Для этого можно отказаться от применения кодов ошибок и перейти к выдаче сообщений об ошибках только тогда, когда они действительно возникают.

```
Option Explicit

Public Function CalculateFuelConsumed(ByVal Duration As Double, _
  ByVal GallonsPerHour As Double) As Double

  If (Duration > 0 And GallonsPerHour > 0) Then
    CalculateFuelConsumed = Duration * GallonsPerHour
  Else
    Call CalculateFuelConsumedError(Duration, GallonsPerHour)
  End If

End Function

Private Sub CalculateFuelConsumedError(ByVal Duration As Double, _
  ByVal GallonsPerHour As Double)

  Const Source As String = "Sheet2.CalculateFuelConsumed"
  Dim Description As String

  Description = "Длительность {"
↳& Duration & "} и Галлонов в час {"
↳& GallonsPerHour & "} должны быть больше 0"

  Call Err.Raise(vbObjectError + 1, Source, Description)
End Sub
Public Sub Test()

  On Error GoTo Catch
    MsgBox CalculateFuelConsumed(-8, 1.5)

  Exit Sub
Catch:
  MsgBox Err.Description, vbCritical

End Sub
```

Функция `CalculateFuelConsumed` выполняет все рассчеты. Если параметры `Duration` и `GallonsPerHour` имеют некорректное значение, сообщение об ошибке выдается через вызов вспомогательной функции `CalculateFuelConsumedError`. Впомогательный метод создает удобно отформатированное описание и выдает сообщение об ошибке при помощи вызова метода `Err.Raise` и передачи необязательного кода ошибки, исходной строки, описания, имени файла справочного руководства и контекста файла справочного руководства в качестве параметров. По соглашению контекстные номера ошибок добавляются в константу `vbObjectError` (как показано ранее). Это позволяет предотвратить пересечение собственных номеров ошибок и номеров ошибок языка VBA.

Для использования метода `CalculateFuelConsumed` создается оператор перехода на метку `On Error GoTo`. Стоит использовать такой способ (`Catch` или `Handle`) постоянно. После этого добавляется вызов метода и оператор выхода из метода: `Exit Sub` для подпрограмм, `Exit Function` для функций или `Exit Properties` для свойств. Наконец, в конце с оптимизмом добавляется метка и код для обработки ошибки. В этом примере пользователь получает сообщение об ошибке, выводимое с помощью оператора `MsgBox`.

Если кого-то интересует, почему этот подход лучше других, можно отметить, что:

- функцию можно использовать, как и предполагалось, для возврата подсчитанного результата;
- не добавляется код ошибки "ничего не делать", который пессимистически использует условный оператор для проверки правильности завершения функции. Можно предполагать, что все работает как надо;
- присутствует страховочная сетка, которая перехватывает любые ошибки, а не только собственные коды ошибок. (Например, можно не проверять неравенство делителя нулю. Если будет выполнено деление на ноль, при этом используется встроенный код ошибки. При пессимистическом подходе эта ошибка прошла бы сквозь систему контроля. Оператор `On Error GoTo` такие ошибки не пропускает);
- объект `Error` передает значение ошибки вместе с сообщением об ошибке. Разработчик избавляется от необходимости создавать смысл ошибки на основе произвольного значения кода ошибки.

Если один стиль лучше с одной стороны, то он лучше и в общем. Выдача сообщений об ошибках лучше возврата кодов ошибок по нескольким несубъективным критериям.

Теперь стоит рассмотреть создание обработчиков ошибок.

Создание обработчиков ошибок

Существует три формы оператора `On Error`. Это оператор перехода на произвольную строку кода `On Error GoTo`. Далее оператор `On Error Resume Next`, обеспечивающий переход на строку сразу после строки, которая привела к появлению ошибки. Кроме этого, еще существует оператор `On Error GoTo 0`, просто сбрасывающий сообщение об ошибке.

Оператор On Error GoTo

Оператор `On Error GoTo` уже использовался вместе с меткой `Catch` в предыдущем разделе. Важно, чтобы метод завершал нормальную работу до метки. Иначе метка и код обработки ошибки будут выполняться даже при нормальной работе. Для завершения ра-

боты подпрограммы применяется оператор `Exit Sub`. Для завершения работы функции — оператор `Exit Function`, а для завершения работы свойства — оператор `Exit Property`. Но что, если код обработки ошибок должен работать всегда? Такое бывает.

Подобный прием называется блоком защиты ресурсов. Компьютеры используют конечное количество сущностей, которые в целом называются ресурсами. Ресурсом может быть подключение к базе данных, файлу или сетевому сокету. Если создаются экземпляры таких ресурсов, то необходимо обеспечить их правильное удаление. Для этого можно воспользоваться идиомой блока защиты ресурсов. Она имеет простой принцип работы: использовать оператор `On Error GoTo` сразу после создания ресурса и намеренно не добавлять оператор завершения процедуры перед соответствующей меткой. Таким образом, код обработки ошибок (в данном случае это код защиты ресурсов) будет выполняться вне зависимости от наличия или отсутствия ошибок. Вот как выглядит реализация идиомы для открытия файла и записи текста в открытый файл. (Не стоит использовать такой способ записи в файлы. Для этого имеет смысл применять объект `FileSystemObject`.)

```
Public Sub ProtectThisResource()

  Open ThisWorkbook.Path & "\dummy.txt" For Output As #1
  On Error GoTo Finally

    Print #1, "Этот файл всегда будет закрыт"

Finally:

  Close #1

  If (Err.Number <> 0) Then MsgBox Err.Description

End Sub
```

> Помните, что в качестве меток могут использоваться произвольные номера строк. Для этого в операторе `On Error GoTo` строку метки можно заменить на число.

Базовая последовательность действий блока защиты ресурсов состоит из создания ресурса, добавления оператора `On Error GoTo`, попытки использования ресурса и очистки (ресурса или ошибки). В этом примере открывается текстовый файл, выполняется оператор `On Error GoTo Finally`, делается попытка использования ресурса и, в любом случае, выполняется очистка. Обратите внимание на отсутствие оператора выхода из процедуры.

Оператор On Error Resume Next

Оператор `On Error Resume Next` может использоваться для игнорирования ошибки и продолжения выполнения со следующего оператора. Этот способ применяется непосредственно перед оператором, который может привести к появлению ошибки (например, перед оператором, результат выполнения которого не особенно важен). Оператор `On Error Resume Next` используется редко, так как не часто создаются операторы, результат выполнения которых не важен. Если результат работы оператора никого не интересует, удалите его.

Операторы `Resume` и `Resume Next` могут использоваться сами по себе. Оператор `Resume` применяется в завершении процедуры обработки ошибки для повтора попытки выполнить оператор. Например, если добавить дополнительный блок обработки ошибок

в метод ProtectThisResource на случай невозможности открыть файл из-за установленного атрибута ReadOnly (Только чтение), то в процессе обработки ошибки можно сбросить этот атрибут и выполнить оператор Resume для повторения попытки открыть файл. Оператор Resume Next передает управление следующему оператору после оператора, который привел к появлению ошибки. Такой оператор используется в ситуациях, когда вызвавшую ошибку строку можно пропустить. В следующем методе показано, как блок защиты ресурсов можно использовать вместе с блоком обработки ошибок, а также как восстанавливать работоспособность после попытки открыть защищенный от записи файл:

```
Public Sub ProtectThisResource()

  Dim FileName As String
  FileName = ThisWorkbook.Path & "\dummy.txt"

    On Error GoTo Catch
  Open FileName For Output As #1

  On Error GoTo Finally
    Print #1, Time & " Этот файл всегда будет закрыт"

Finally:

  Close #1
  If (Err.Number <> 0) Then MsgBox Err.Description
  Exit Sub

Catch:
  If (Err.Number = 75) Then
    Call SetAttr(FileName, vbNormal)
    Resume
  End If

End Sub
```

Первый оператор On Error GoTo Catch передает управление обработчику защищенных от записи файлов, позволяющему открыть файл еще раз после сброса атрибутов. Второй оператор On Error GoTo обеспечивает закрытие файла после завершения работы процедуры.

Пример метода может показаться слишком сложным и стоит обратить внимание, что ошибки могут возникнуть и в других местах. Что, если поврежден диск? Что, если недостаточно памяти для загрузки слишком большого файла? Что, если файл заблокирован другим приложением? Возможны любые ошибки. Именно поэтому сложно написать стабильную программу, а также предусмотреть все возможные варианты и все условия. Программист должен субъективно оценить возможные неприятности и попытаться обойти их. Этот процесс займет некоторое время, но в итоге он все равно завершится и код будет передан пользователям. Некоторые специалисты говорят, что создание программного обеспечения компьютеров является самым сложным видом деятельности. С этим нельзя не согласиться.

На этом этапе уже были показаны некоторые ошибки, которые могут возникать и в простом коде. Теперь представьте себе создание 10 или 20 миллионов строк защищенного кода, лежащих в основе Windows или Windows NT. На компанию Microsoft оказывается постоянное давление с целью стимулировать создание более защищенного кода Windows, но есть очень умные люди, которые наслаждаются процессом обнаружения дыр в Windows. Удивительно то, что это происходит не слишком часто.

Оператор On Error GoTo 0

Оператор `On Error GoTo 0` отключает обработчики ошибок в текущей процедуре. Это еще один оператор, который используется не очень часто. Но иногда он встречается. Его можно воспринимать, как выключатель обработчиков ошибок на уровне процедуры.

Использование объекта Err

Объект `Err` содержит информацию о самой последней ошибке. В объекте хранится код ошибки, источник ошибки, описание ошибки и ссылка на документ справочного руководства, если таковое существует.

Объект `Err` является экземпляром шаблона Singleton. Это значит, что такой объект существует в единственном экземпляре в пределах приложения. Так как это экземпляр класса, он имеет свойства и методы. Для выдачи сообщения об ошибке можно воспользоваться методом `Err.Raise`, а для очистки сообщения — методом `Err.Clear`. Оставшиеся свойства (кроме одного) уже рассматривались. Они используются для инициализации ошибок. Последним нерассмотренным свойством является `LastDllError`. Это свойство возвращает значение типа `Hresult`, которое обычно возвращается из библиотек DLL. Это свойство необходимо использовать при вызове методов во внешних библиотеках DLL, например в библиотеках Windows API.

Создание обвязки

Перед тем как перейти к обсуждению журнала событий, стоит обратить внимание на то, где и когда необходимо создавать тестовый код. В данном случае используется метод обвязки (scaffolding). Если программирование похоже на добавление рассказов в общую структуру, то создание обвязки предполагает добавление текста к каждому рассказу. Создание обвязки позволяет убедиться, что новый код является изолированным и не добавляет ошибки в уже существующий проверенный код.

Например, модуль `DebugTools` был создан для повторного использования в других приложениях. Данный код должен быть изолирован, но в этом не будет уверенности, пока не протестирован отладочный код. Тогда необходимо добавить по одному тестовому методу для каждого открытого метода в модуле `DebugTools`, вызывающего код `Trace`, `Assert` и `Trap`. Это позволяет убедиться, что выполнение кода приводит к получению ожидаемого результата. (Было бы просто смешно, если бы отладочный код становился причиной появления ошибок.) Вот полный листинг модуля `DebugTools` вместе с обвязкой тестового кода, выделенной полужирным шрифтом.

```
Option Explicit

#Const Tracing = True
#Const Debugging = True
#Const UseEventLog = False

Public Sub Trap(ByVal Source As String, _
  ByVal Message As String)

#If Debugging Then

 Call DebugPrint(Source, Message)
 Stop
```

```
#End If

End Sub
Private Sub DebugPrint(ByVal Source As String, _
  ByVal Message As String)

#If UseEventLog Then

#Else
  Debug.Print "Источник: " & Source & " Сообщение: " & Message
#End If

End Sub
Public Sub Assert(ByVal Test As Boolean, _
  ByVal Source As String, ByVal Message As String)

#If Debugging Then
  If (Not Test) Then
    Call DebugPrint(Source, Message)
    Debug.Assert False
  End If
#End If

End Sub
Public Sub Trace(ByVal Source As String, _
  ByVal Message As String)

#If Tracing Then
  Call DebugPrint(Source, Message)
#End If

End Sub
Public Sub TraceParams(ByVal Source As String, _
  ParamArray Values() As Variant)

  Dim Message As String
  Dim I As Integer

  For I = LBound(Values) To UBound(Values)
    Message = Message & " " & Values(I)
  Next I

  Call Trace(Source, Message)

End Sub
#If Debugging Then

Public Sub TrapTest()
 Call Trap("Sheet1.CallTrap", "Test Trap")
End Sub
Public Sub AssertTest()
   Call Assert(False, "AssertTest", "Assertion Failure")
End Sub
Public Sub TraceTest()
   Call Trace("Sheet1.TraceAssert", "Trace Test")
End Sub
Public Sub TestSuite()
   ' Закомментировать после завершения всех тестов
   TrapTest
   AssertTest
```

```
   TraceTest
End Sub
#End If
```

Тестовый код доступен только при установке переменной `Debugging` в значение `True`. Метод `TestSuite` вызывает тестовые методы (`TrapTest`, `AssertTest` и `TraceTest`), которые, в свою очередь, вызывают отладочные методы, а результат вызова можно пронаблюдать в окне Immediate (Проверка). Построчное выполнение кода позволяет убедиться в получении необходимых результатов перед передачей кода другим разработчикам.

В создании обвязки нет ничего сложного. Но это необходимо делать в процессе написания кода. Тестируйте каждый уровень в процессе увеличения сложности, а не все сразу после того, как написание основного кода завершено. Создание нескольких уровней тестов является настолько же важным, как и создание нескольких уровней кода. Каждый фрагмент будет базироваться на надежной основе.

Запись в журнал событий

Журнал событий является системным ресурсом. Он достаточно важен и ценен, чтобы быть неотъемлемой частью новой инфраструктуры .NET от компании Microsoft. (Инфраструктура .NET предназначена для использования в языках программирования Visual Basic, C#, C++ и множестве других языков программирования.) Это настолько важный ресурс, что компания Microsoft включила его в Exception Management Application Block (EMAB) и Enterprise Instrumentation Framework (EIF). Дополнительная информация о EMAB и EIF доступна в сети Internet. В отличие от журнала событий эти механизмы не доступны для непосредственного использования, поэтому ниже рассматривается только журнал событий.

Журнал событий является локальной системной службой, выступающей в роли хранилища информации о состоянии приложений, безопасности и компьютере в целом. Журнал событий можно использовать в процессе диагностики, так как его содержимое сохраняется между сеансами работы приложения. Следовательно, сообщения в журнале событий не исчезнут даже в случае аварийного отказа приложения. Эта информация позволит определить причину отказа. Другими словами, окно Immediate (Проверка) — это хорошо, но журнал событий доступен всегда.

Следующий код доступен в файле `EventLog.bas`. В коде используется шесть методов Windows API, позволяющих подключиться к журналу событий Windows и упрощающих запись сообщений об ошибках до использования единственного метода `WriteEntry`. В этом случае не показана вся гибкость возможностей журнала событий, но на данном этапе журнал событий будет использоваться только для записи сообщений об ошибках:

```
Option Explicit
Private Const GMEM_ZEROINIT = &H40 ' Initializes memory to 0

Private Const EVENTLOG_ERROR_TYPE = 1
Private Const EVENTLOG_WARNING_TYPE = 2
Private Const EVENTLOG_INFORMATION_TYPE = 4

Declare Function RegisterEventSource Lib "advapi32.dll" Alias
↳ "RegisterEventSourceA" (ByVal MachineName As String,
↳ ByVal Source As String) As Long

Declare Function ReportEvent Lib "advapi32.dll"
```

```
↳ Alias "ReportEventA" (ByVal Handle As Long,
↳ ByVal EventType As Integer, ByVal Category As Integer,
↳ ByVal EventID As Long, ByVal UserId As Any,
↳ ByVal StringCount As Integer, ByVal DataSize As Long,
↳ Text As Long, RawData As Any) As Boolean

Declare Function DeregisterEventSource Lib "advapi32.dll" (ByVal
↳ Handle As Long) As Long

Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" (ByVal
↳ Destination As Any, ByVal Source As Any, ByVal Length As Long)

Declare Function GlobalAlloc Lib "kernel32" (ByVal Flags As Long,
↳ ByVal Length As Long) As Long

Declare Function GlobalFree Lib "kernel32" (ByVal
↳ hMem As Long) As Long

Public Sub WriteEntry(ByVal Message As String)

  Dim Handle As Long
  Dim EventSource As Long

On Error GoTo Finally
  Handle = GlobalAlloc(GMEM_ZEROINIT, Len(Message) + 1)
  Call CopyMemory(Handle, Message, Len(Message) + 1)
  EventSource = OpenEventSource("vbruntime")

  Call ReportEvent(EventSource, EVENTLOG_ERROR_TYPE, _
    0, 1, 0&, 1, 0, Handle, 0)

Finally:
  If (Handle <> 0) Then Call GlobalFree(Handle)
  If (EventSource <> 0) Then CloseEventSource (EventSource)
End Sub
Public Function OpenEventSource(ByVal Source As String) As Long
  ' Использовать локальный компьютер
  OpenEventSource = RegisterEventSource(".", Source)
End Function

Public Sub CloseEventSource(ByVal EventSource As Long)
  Call DeregisterEventSource(EventSource)
End Sub
Sub LogEventTest()

  Call WriteEntry("Это тест!")
End Sub
```

После добавления модуля `EventLog` можно изменить значение переменной `UseEventLog` на `True` и вызывать метод `WriteEntry` для записи информации об ошибках в журнал событий. Для просмотра журнала приложений воспользуйтесь утилитой Просмотр событий (Event Viewer) (выберите команду Пуск⇨Выполнить (Start⇨Run) и введите `Eventvwr.msc`). Записи в журнале будут иметь источник `vbruntime`, как показано на рис. 7.6.

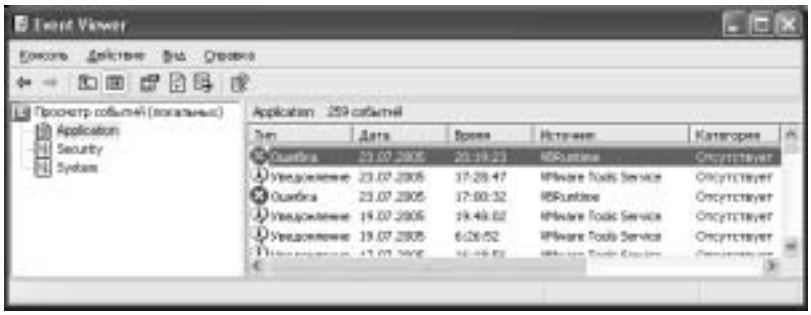

Рис. 7.6. Журнал событий. Записи об ошибках

Модуль `EventLog.bas` можно рассматривать как приманку для главы 16. (Невозможно уместить все в одной главе, поэтому использование Windows API рассматривается более подробно в главе 16, "Программирование с помощью Windows API".)

Резюме

Для того чтобы что-то сделать хорошо, необходимо просто упорядочить хаос. Для того чтобы что-то сделать хорошо и быстро, необходима практика, привычка и хороший набор инструментов. В этой главе было показано, как создавать полезные отладочные инструменты и инструменты тестирования на основе объекта `Debug`, оператора `Stop` и журнала событий Windows. Выработанная привычка создавать код с помощью таких отладочных инструментов позволяет сразу получать качественный код и делать это быстро.

В этой главе рассматривались методы `Assert`, `Trace` и `Trap`. Кроме этого, было показано, как использовать функции Windows API для записи ошибок в журнал событий. Большинство неясных моментов применения Windows API будут рассмотрены в главе 16.