StateCoder™

Version 2.0

for Visual Studio .NET

by

Desaware, Inc.

Rev 2.0.0 (12/05)

Desaware, Inc.
Software License

Please read this agreement. If you do not agree to the terms of this license, promptly return the product and all accompanying items to the place from which you obtained them.

This software is protected by United States copyright laws and international treaty provisions.

This program will be licensed to you for use only on a single computer. If you wish to install it on additional computers, you must purchase additional software licenses. You may (and should) make archival copies of the software for backup purposes.

You may transfer this software and license as long as you include this license, the software and all other materials and retain no copies, and the recipient agrees to the terms of this agreement.

You may not make copies of this software for other people. Companies or schools interested in multiple copy licenses or site licenses should contact Desaware, Inc. directly at (408) 404-4760.

Should your intent be to purchase this product for use in developing a compiled .NET program of the following types: Windows application (.exe), Windows service (.exe), ASP.NET web application (.dll) or Web service (.dll), you may create runtime license certificates and distribute the StateCoder redistributable files without paying additional royalties. Review the listing of which files (located below) that can be distributed and or modified. If Desaware files are included in your executable program, you must include a valid copyright notice on all copies of the program. This can be either your own copyright notice, or "Copyright © 2005 Desaware, Inc. All rights reserved."

Should your intent be to purchase this product for use in developing a compiled .NET components of the following types: Windows control (.dll), Web control (.dll) or Component (.dll) or Class library (.dll), you must purchase an embedded software license from Desaware before you may distribute your component.

You have a royalty-free right to incorporate any of the sample code provided into your own applications with the stipulation that you agree that Desaware, Inc. has no warranty, obligation or liability, real or implied, for its performance.

Desaware.StateCoder11.dll and DesawareStateCoder20.dll: You may include with your program a copy of these files under the terms in the preceding paragraphs.

StateCoder Source Files: Source code for portions of StateCoder are included for educational purposes only. You may use this source code in your own applications only if they provide primary and significant functionality beyond that included in the StateCoder package. You may not use this source code to develop or distribute components and tools that provide functionality similar to all or part of the functionality provided by any of the components or tools included in the StateCoder package.

Please consult the on-line Help file under the topic File Descriptions for additional information.

Limited Warranty

Desaware, Inc. warrants the physical CD and physical documentation enclosed herein to be free of defects in materials and workmanship for a period of sixty days from the date of purchase.

The entire and exclusive liability and remedy for breach of this Limited Warranty shall be limited to replacement of defective CD(s) or documentation and shall not include or extend to any claim for or right to recover any other damages, including but not limited to, loss of profit, data or use of the software, or special, incidental or consequential damages or other similar claims, even if Desaware, Inc. has been specifically advised of the possibility of such damages. In no event will Desaware, Inc.'s liability for any damages to you or any other person ever exceed the suggested list price or actual price paid for the license to use the software, regardless of any form of the claim.

DESAWARE, INC. SPECIFICALLY DISCLAIMS ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Specifically, Desaware, Inc. makes no representation or warranty that the software is fit for any particular purpose and any implied warranty of merchantability is limited to the sixty-day duration of the Limited Warranty covering the physical CD and documentation only (not the software) and is otherwise expressly and specifically disclaimed.

This limited warranty gives you specific legal rights. You may have others, which vary from state to state.

This License and Limited Warranty shall be construed, interpreted and governed by the laws of the State of California, and any action hereunder shall be brought only in California. If any provision is found void, invalid or unenforceable it will not affect the validity of the balance of this License and Limited Warranty, which shall remain valid and enforceable according to its terms.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software - Restricted Rights at 48 CFR 52.227-19, as applicable. Contractor/Manufacturer is Desaware, Inc., 3510 Charter Park Drive, Suite 48, San Jose, California 95125.

Table Of Contents

Introduction

Introducing Desaware's StateCoder presents to us a unique problem. You, the reader, may come from many different backgrounds.

Some of you, upon hearing the words "Finite State Machine" will immediately recognize the term as describing a design pattern with which you are not only familiar, but use routinely in virtually every program you write. For you, the introduction needs to explain why Desaware's StateCoder is a much better way for you to create state machines in .NET. It will save you time, offer improved reliability in areas ranging from thread safety to long term supportability, improve your application's efficiency and performance – in short, reduce your costs of developing high quality .NET software.

But I realize that many of you, especially those of you who are self-taught, may not be familiar with state machines at all. It's a subject that tends to be covered mostly in academic press and articles rather than in the more practical form needed by most software developers. Many of you have heard of state diagrams, and perhaps seen them used in UML documents, but have not necessarily translated them into real code. For you, the introduction really needs to teach you about state machines. You see, you're already using them, whether you realize it or not. But if you take the time to learn more about state machines, you'll find the benefits to your code will be enormous – because state machines are applicable everywhere. Once you learn about state machines, I'll show you why Desaware's StateCoder is the best way to create and use state machines in .NET.

So here's the deal.

For those of you who need to learn more about State Machines, in the next section I'll go into an in-depth, practical and very non-academic tutorial on state machines and the design patterns that go with them. But first, I'd like to provide a feature list of Desaware's StateCoder for those who are

familiar with State Machines. Don't worry if you don't understand these features now – by the time you finish the tutorial, you'll understand them all:

One last thought before going into the feature list: While we did develop StateCoder to help .NET programmers create better .NET programs, we also created it as the base framework on which we'll be building additional .NET products and services. So when we talk about the benefits of StateCoder, know that our own developers are our first (though not necessarily most important) customers.

StateCoder Features

StateCoder is a .NET namespace that is designed to make it easy to create and support powerful State Machines in .NET using Visual Basic .NET, C# and other .NET Languages.

With Desaware's StateCoder, you will create .NET code that is:

- More reliable
- Easier and cheaper to test
- Easier and cheaper to support, understand and to modify safely
- More immune to threading synchronization problems
- More efficient – uses fewer system resources
- Cheaper and faster to develop.

Typical Applications for Desaware's StateCoder include:

- Management of asynchronous operations. Especially useful for managing large numbers or varying numbers of asynchronous operations.

- Dramatically reduce the number of threads needed to perform background operations (especially useful for objects that support multiple clients)
- Selective use of between-request processing for ASP.NET and Web Services can dramatically improve user response, while using minimal server resources.
- Encapsulating sequences of asynchronous operations (wrapping them into a single event or wait operation).
- Protocol Implementation. Internet or other protocols are almost always based on state machines.
- Data format conversions. State machines can be used to parse incoming data and perform operations based on the results, including generating output data in a different format.
- Do-it-yourself transactioning. State machines can define transaction based objects that don't use COM+/MTS.

The classes of the Desaware.StateCoder namespace form a framework that offers the following features:

- State machines can be defined statically using attributes or dynamically.
- Flexible thread control - from using a single thread to support unlimited state machines to assigning each state machine its own thread.
- State machines can be assigned to web applications and services, running in the background between requests.
- Largely self-synchronizing – when used correctly, eliminates the chance of data corruption, race conditions and deadlocks (and

reduces the chance of these things happening even when used incorrectly).

- Complete flexibility with regards to message sources. Base message sources include:
    - Generic source for custom message sources.
    - IAsyncResult based source for asynchronous operations.
    - Stream based message source with predefined or custom parsing into messages.
- Message sources can be built on our base types (one or two overrides and you're done), or created from scratch.
- Messages can be of any .NET type.
- Multiple message sources are allowed.
- Message sources can be redefined for each state in the state machine.
- Message sources can include timeouts.
- Exceptions can act as message sources.
- State machines can act as messages sources to other state machines.
- State machines can invoke other state machines.
- State machines can be run by the StateCoder framework, or create your own framework for running a state machine – or run it synchronously.
- Fully resource driven – easy to localize.
- Per developer licensing makes it inexpensive to incorporate StateCoder into your own applications and services.

New Features for Version 2.0

Version 2.0 of StateCoder incorporates the following new features:

- .NET 2.0 Framework components.
- New Generic based State class.
- New QueuedStream class.
- Updates for compatibility with .NET 2.0 and ASP .NET 2.0.

This edition of StateCoder includes two StateCoder components:

- Desaware.StateCoder11.dll for use with the .NET 1.1 framework
- Desaware.StateCoder20.dll for use with the .NET 2.0 framework

The two components are almost identical – the main difference being that the .NET 2.0 version includes generic forms of the State and DynamicState classes.

We no longer recommend installing StateCoder in the GAC (in order to simplify versioning and distribution), though you may continue to do so.

The version 2.0 components are source compatible with your existing StateCoder components but are not drop-in replacements. You must rebuild your application to use them.

What are State Machines, and Why Should You Care?

A State Machine, in the context of software, is a way of organizing the operations that take place in an application. The idea is that your program exists in a finite number of possible states, and that something happens to move your program from one state to the next.

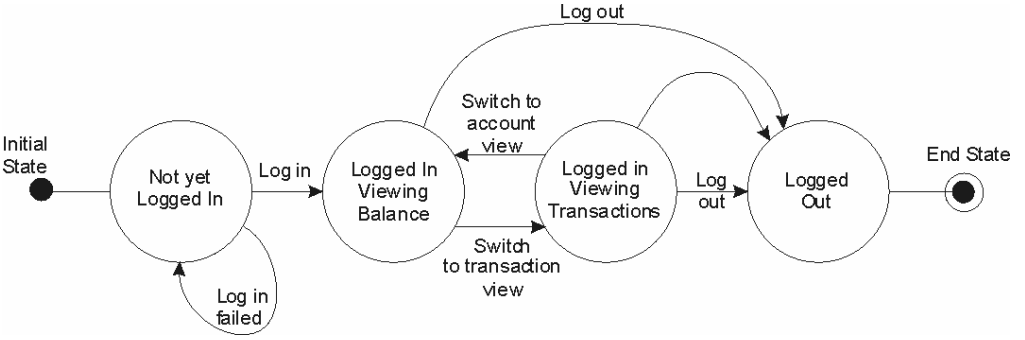
For example: You might have a web application that allows you to log in and view your account information. This could be divided into the following states:

- Not yet logged in.
    - Action: Display the login page
- Logged in and viewing the account balance
    - Display the account balance
    - Display logout button, and view transaction button.
- Logged in and viewing recent transactions
    - Display recent transactions
    - Display logout button, and view account balance button.
- Logged out
    - Display “Goodbye page”

At any given time, the application exists in one of these four states. In each state there exist a limited number of possible events (typically called “messages” when discussing state machines). For example, during the “Logged in” state, the page displays text boxes for the user name and password and a “login” button. If the user enters a correct login, the application switches to the “Logged in and viewing the account balance”

state. If the login fails, the user sees an error message and the application remains in the “Not logged in state”.

State machines are frequently described using State Diagrams (or State Transition Diagrams – STD) that look something like this:



The initial state is where the state machine starts (the circle marked Initial state, and the one marked End State are not states themselves – rather pointers to the actual initial and end states). Each state is represented by a circle. Each arrow represents a message that can be received by the state machine (often triggered by an event) that can cause the machine to change states. Thus in this example, a successful log-in moves the application into the account viewing state.

Why are State Machines Important?

To understand why state machines are so important, remember that much of what we do as software developers consists of managing complexity. We implement very complex applications and algorithms by breaking them up into smaller manageable tasks.

One of the key purposes of object oriented programs (OOP) is to help manage complexity. By using information hiding within objects (implementing private functions and variables) and defining a limited number of public methods and properties, you are able to deal with an object as a single indivisible entity. Once the object is created and

implemented, you no longer have to worry about how it works or the possibility of accidentally modifying one of its internal data variables. Thus Object Oriented Programming inevitably results in simpler programs – programs that are easier to understand and support.

State machines serve the same purpose, but on an architectural level. As part of the development of a state machine you define all of the valid events that may occur during that state. For each event you define an action and a state transition (which includes the possibility of remaining in the same state). You might also define an action for all invalid states.

What does this accomplish?

- First, it makes the program far easier to modify. Adding new features might consist of adding new states. By clearly defining the events that can bring you to that state and events for that state you at the same time define the code that needs to be modified. Code that is not involved with that state can be safely ignored.
- It becomes dramatically easier to test programs implemented as state machines. Why? Because it is possible to break down the testing process into testing of individual states. If you test each state for all of its possible events (a reasonable task), you can go a long way to eliminating bugs in your program. True, there may remain subtle bugs due to the interaction of states (especially if they share any data), and your state machine may itself have design flaws (say, forgetting a particular event), but the results of this approach will always result in a higher quality program than otherwise.
- Using state machines also demands that you spend some time designing them before you start coding. And let's face it, design time is something that developers often skimp on, especially in the face of deadline pressures.

Here's another way of looking at it.

Figure 1 shows how object oriented programming reduces complexity by reducing the number of functions and variables you need to deal with at a given level of your program. In this illustration you can see on the left side a large number of variables and functions that might appear in a non OOP program. When using OOP the variables and functions are hidden inside of objects. These objects expose a limited number of methods that provide a high level encapsulation of the enclosed variables and functions. As a result, once you've implemented these objects, instead of having to worry about a large number of variables and functions (and their interactions), you need only concern yourself with a small number of objects and their methods. Fewer items to work with results in reduced complexity, increased reliability, and overall lower software development costs.

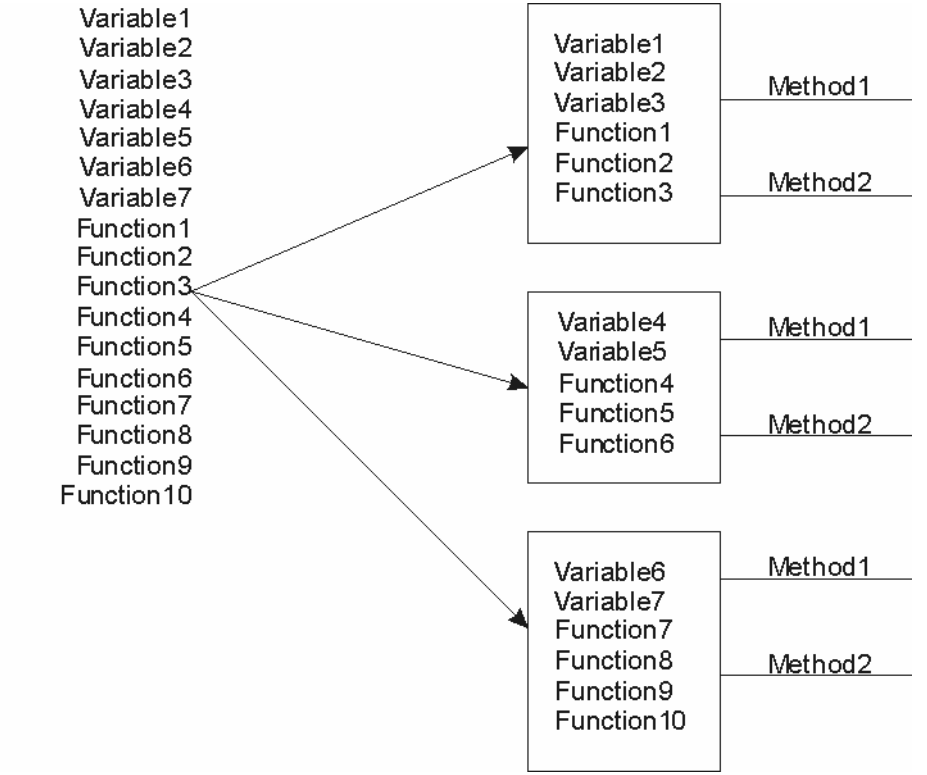


Figure 1 – Object oriented programming.

Now consider figure 2. On the left side, instead of lists of functions and variables you can see a list of events. These are possible inputs to your program. These can be in the form of user actions, data received from a network, data read from a disk or other source and even results of an operation or exceptions that occur while a program is running - basically anything that can represent input to your program.

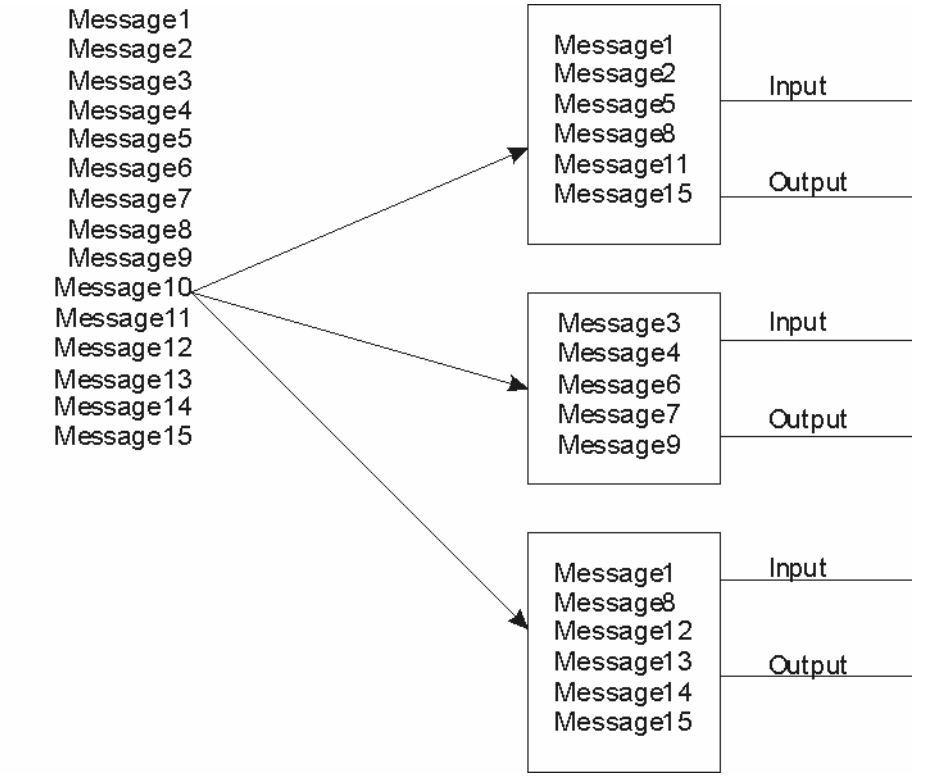
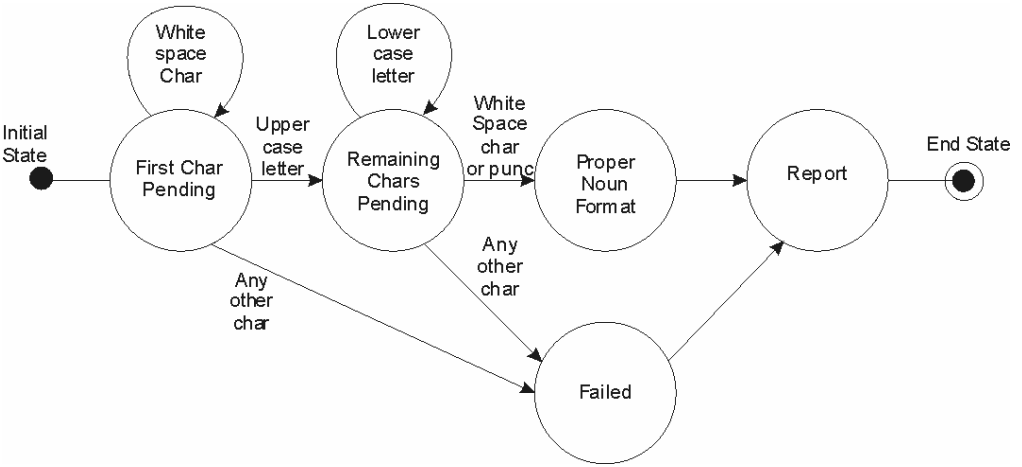


Figure 2 – State machine programming.

If your code has to consider all possible inputs at all times, the complexity of the any non-trivial program would be impossible to deal with. Fortunately, this is rarely the case – programs naturally deal with certain events at certain times. A function that reads data from disk rarely worries about user input. Yet at the same time, that function that reads data from disk may receive unanticipated input – a user abort or disk error, and the failure to deal with unanticipated input is a key source of bugs and instability in software.

A state machine serves to divide an application's life into a series of states, each of which has a set of acceptable input. Once in a given state, you need only concern yourself with valid inputs. Invalid inputs are by definition errors that can be trapped and handled. Just as OOP simplifies a program by reducing the number of variables and functions you need to deal with, state machines simplify a program by reducing the number of inputs you need to deal with. Collapsing a given set of inputs into a state machine that has a set start point and end point, and can be dealt with as a single entity, just as an object can deal with a group of variables and functions as a single entity.

State machines work at multiple levels. Consider the following state machine:



This represents a state machine that processes incoming characters to look for words in the format of a proper noun (i.e., the first character is upper case, all other characters are lower case.

The first state handles three possible messages. A white space character (such as space or tab) indicates the word has not yet started, so the machine remains in the same state. An upper case character means that the word has started, so the machine transitions into the second state. Any other character represents a failure, so the machine moves into the Failed state. Once in the second state, all subsequent lower case letters indicate continuation of the word, so the machine remains in the same state. A white space or legal punctuation indicates the end of the word, which moves the machine into the "success" state. any other character again moves the machine into the failed state.

Do people use state machines to process text in this manner? Absolutely. In fact, the .NET framework includes a namespace called System.Text.RegularExpressions whose sole purpose is the processing of text using state machines that are defined using a special Regular Expression Language! For an in-depth tutorial and reference of this language (that is considerably easier to follow than Microsoft's documentation) refer to the E-book "Regular Expression in .NET" which can be purchased from http://www.desaware.com/products/books/net/regexpressions/index.aspx .

State Machine Design Patterns – or How State Machines can help you avoid creating bad .NET code.

"Design patterns" is one of those terms that has received a great deal of attention lately and can be intimidating to those who don't realize that the idea behind them is very simple. A design pattern is just a common way of doing something.

For example: let's say you want to swap two elements in an array. The following pseudocode[1] demonstrates a design pattern that performs this task:

```
Temp = A(n)
A(n) = A(m)
A(m) = Temp
```

Unless a language contains a function designed to swap two variable, you'll see something that looks more or less like this code anywhere that two array elements need to be swapped. This code can thus be thought of as a design pattern for swapping elements in an array.

Design patterns enable us to look at common solutions (both good and bad) for various problems.

Now, state machines are applicable in many different scenarios. There are many problems that can be solved both with and without clearly defined state machines, where the state machine based design pattern will improve

[1] Pseudocode is "fake" code. It's code that is in no particular language, and can't actually be compiled and run. It's a mixture of descriptive text and something that looks like code that anyone with a familiarity with any block structured language (such as C#, VB, C++) can understand.

the code[2]. But there are a number of areas in .NET where state machine based design patterns are vastly better than any other approach. They are:

- Performing long synchronous operations
- Any time you handle asynchronous operations
- Any time you use multithreading
- When implementing a protocol based application

In this section, you'll see common but flawed design patterns for the first two of these scenarios, and how you can use a state machine based design pattern to eliminate those flaws. The third scenario will become clear as you read the first two. The fourth scenario won't be demonstrated because it's obvious – you can't implement a non-trivial protocol without a state machine – so any implementation that doesn't use a clearly defined (and properly designed) state machine is bound to be less reliable than an implementation that is coded haphazardly.

There are many applications for state machines that fall outside of these design patterns, some of which can be found in articles on our web site, but these four areas tend to benefit most from formal implementation in state machines (largely because implementations that don't include formal implementations of state machines tend to suffer the most problems).

Long Synchronous operations

One common problem in single threaded applications relates to performing a sequence of time consuming synchronous operations. Let's say you have three long operations to perform in sequence. These can be CPU intensive operations, or operations such as service or network requests or database queries that are synchronous (i.e., you must wait until

[2] I say clearly defined, because in many cases your code is acting as a state machine even if you haven't thought about it.

the operation concludes in order to continue). The obvious design pattern is:

```
LongOperation1( )
LongOperation2( )
LongOperation3( )
```

Or the closely related pattern

```
Do
      LongOperation
Loop While …
```

The problem with this approach in a single threaded application is that while you are waiting for this sequence to conclude, the rest of your application is frozen. One design pattern I've seen in all too many Visual Basic programs tries to alleviate the problem as follows:

```
LongOperation1( )
Application.DoEvents[3]( )
LongOperation2( )
Application.DoEvents( )
LongOperation3( )
```

Or

```
Do
      LongOperation
      Application.DoEvents( )
Loop While …
```

The DoEvents allows temporary event processing to occur (it's the equivalent of a PeekMessage call for those of you from the C++ world). This is a terrible design pattern. First, the degree to which it actually helps depends on the length of the long operations. This often results in "jumpy" form behavior. Worse, the DoEvents command adds the possibility of reentrancy – during the event the entire gamut of input messages becomes

[3] The VB6 DoEvents command has its equivalent in the .NET System.Windows.Forms.Application object.

possible and must be dealt with – otherwise you might find yourself reentering this sequence – a potentially fatal problem. It is the exact opposite of simplification.

A better design pattern is to turn this into a state machine such as the one shown in the following pseudocode.

```
Enum StateVariable
      0 = DoLongOperation1
      1 = DoLongOperation2
      2 = DoLongOperation3
End Enum

On Timer
      Based on StateVariable
      Case 0:     LongOperation1 ( )
                  StateVariable = 1
      Case 1:     LongOperation2 ( )
                  StateVariable = 2
      Case 2:     LongOperation3( )
End
```

For a loop, you would have a single state, and during the state would determine (based on a counter) if the state machine should terminate or continue.

Now, this pattern is better. You'll still get jumpy performance because you are using a single thread, but you no longer have the problem of reentrancy. Yes, input may come in between states, but you can easily detect the current status of your state machine and deal with the input accordingly.

With .NET, it becomes easy to create multithreaded applications, in which case the following pattern becomes possible.

In main thread:

```
Create LongOperationThread( LongOperationThread_Start
)
```

In the launched thread

```
LongOperationThread_Start
      LongOperation1( )
      LongOperation2( )
      LongOperation3( )
End
```

You can detect whether the operations are complete by checking the thread object or waiting for the thread to terminate using the Join method.

This is a very reasonable pattern for this simple case. Problems occur in three cases:

- What if this is a business object that has to support multiple clients. Creating a new thread for each client or on each call is very expensive in terms of system resources and can seriously impair performance.
- What if its not a simple sequence of long operations, but rather a series of long operations, where the choice of the operation to perform depends on the results of the previous operation?
- If the long operations share ANY data with the main thread, you run into the possibility of memory corruption due to a failure to synchronize access to the data.

In the first case, the answer is to use a thread pool to perform the desired operations. Each client uses a thread from the pool if one is available, waiting if necessary for a thread to become available.

In the second case, the obvious answer is to implement a state machine instead of a simple sequence. After each operation returns you can choose the next state based on the result of the previous operation.

The third case is by far the most serious. Chapter 7 of Dan Appleman's book "Moving to VB.NET: Strategies, Concepts and Code" includes an in-depth explanation of the sources of these synchronization problems and the risks involved. How big are the risks? The book demonstrates an easy

to overlook problem in a financial application which causes an error on the average of once every 50 million operations. The error causes an arbitrary amount of money to appear or vanish. Obviously, it is not feasible to test for errors that occur so rarely, yet the cost of these errors can be virtually unlimited. Careful design is essential when creating multithreaded applications.

Multithreading synchronization problems can occur any time data is shared among threads. It is especially serious in .NET because the .NET framework itself is not by and large thread safe. Thus it is essential that programmers use great care when deciding to launch additional threads in their applications – especially if they do not have experience designing multithreaded applications.

Desaware's StateCoder is ideal for implementing this design pattern. It addresses all three issues:

1. State machines implemented using StateCoder can be run in their own thread, or in a thread pool – as you prefer.
2. It is trivial to change the sequence of long operations into states. The transition from one state to another is a single method call, and the choice of state can be based on the results of the previous operation.
3. With Desaware's StateCoder, each state is an object that is not accessible from the main operation other than through messages. The architecture makes it easy to isolate data from the main thread (in fact, you have to go out of your way to share data). More important, each state machine runs in a single thread, and incoming messages are synchronized to that thread. You'll read more about how this works later. But in a nutshell – it provides much of the thread safety with which Visual Basic programmers are familiar, but unlike Visual Basic 6, does not prevent you from bypassing the protection an performing your own synchronization.

How this is accomplished will be discussed later. Meanwhile, the StateCoder design pattern for this might be as follows:

In the main thread:

```
Create LongOperationStateMachine object.
LongOperationStateMachine.Start( )
```

You can detect whether the operations are complete by polling the state machine object, waiting for it (using a wait operation), or watching for an event. The state machine object itself contains objects and code that implement the state machine, but the details of that implementation are hidden at this higher level of abstraction. The result is a dramatic simplification in your program (especially in the case of more complex state machines).

Long operations pose additional challenges if you want them to be interruptible. For further discussion of design patterns related to interruptible long operations, refer to our web site at http://www.desaware.com/tech/statemachinelongop.aspx .

Asynchronous Operations

Regardless of whether a long operation takes place in a background thread or a foreground thread, sequentially or as part of a state machine, that long operation will tie up a thread for some period of time. Many such operations provide the ability to perform the operation asynchronously. The design pattern for an asynchronous operation in .NET is shown in the following pseudocode:

The main thread starts the asynchronous operation:

```
AsyncResultVariable =
BeginOperation(DelegateToCallOnCompletion)
```

This function return immediately. When the operation is complete, the completion function is called:

```
CallOnCompletion(AsyncResultVariable)
      EndOperation(AsyncResultVariable)
End
```

A state machine to perform the three long operations can be implemented using a series of events as follows:

```
AsyncResultVariable =
      BeginOperation(
            DelegateToCompletionLongOperation1)

CallOnCompletionLongOperation1(AsyncResultVariable)
      EndOperation(AsyncResultVariable)
      AsyncResultVariable =
      BeginLongOperation2(
            DelegateToCompletionLongOperation2)
End

CallOnCompletionLongOperation2(AsyncResultVariable)
      EndOperation(AsyncResultVariable)
      AsyncResultVariable =
      BeginLongOperation3(
            DelegateToCompletionLongOperation2)
End

CallOnCompletionLongOperation3(AsyncResultVariable)
      EndOperation(AsyncResultVariable)
End
```

You can, in fact, build a complex state machine simply by choosing which asynchronous operation to start in each event.

If you were to look at a StateCoder implementation of a similar state machine, you might find a similar pattern inside the state machine itself. StateCoder provides full support for asynchronous operations, and in fact encourages their use. The difference being that while the design pattern might be used inside the state machine, the high level design pattern for using the state machine would again look like this:

```
Create LongOperationStateMachine object.
LongOperationStateMachine.Start( )
```

Isolation of the state machine implementation from the rest of your code is a key feature of StateCoder.

What you don't want to do is implement this design pattern:

```
AsyncResultVariable =
      BeginOperation(
            DelegateToCompletionLongOperation1)
Do
      DoEvents
Loop until CompletionFlag1

AsyncResultVariable =
      BeginOperation(
            DelegateToCompletionLongOperation2)
Do
      DoEvents
Loop until CompletionFlag2

AsyncResultVariable =
      BeginOperation(
            DelegateToCompletionLongOperation3)
Do
      DoEvents
Loop until CompletionFlag3
Continue running...

CallOnCompletionLongOperation1(AsyncResultVariable)
      EndOperation(AsyncResultVariable)
      CompletionFlag1 = True
End

CallOnCompletionLongOperation2(AsyncResultVariable)
      EndOperation(AsyncResultVariable)
      CompletionFlag2 = True
End

CallOnCompletionLongOperation3(AsyncResultVariable)
      EndOperation(AsyncResultVariable)
      CompletionFlag3 = True
```

```
End
```

The programmer implementing this code is trying to run asynchronous operations synchronously. At first glance, this code looks truly terrible. But in fact, it is truly terrible. It does avoid the problem of freezing the main thread, but opens the door to the reentrancy problems described earlier. Any time you see this design pattern, your code is begging for a redesign into a state machine.

Why Desaware's StateCoder is the best way to implement State Machines in .NET

Visual Basic developers know that components and products from Desaware tend to be unique.

- We offered tools for subclassing and hooks before most VB programmers even knew what subclassing was (SpyWorks – now for .NET also).
- We offered components for using OLE Structured Storage from VB when most VB programmers didn't know what OLE Structured storage is, and why it's a great way to store data in certain applications (StorageTools – now for .NET also).
- We figured out a way to make world class NT Services using VB – so easily that many of our customers switched from C++ to VB just to use this tool (honest! – Desaware's NT Service Toolkit – also the best way to create Services for .NET).
- We offered tools for detecting component conflicts and resolving "DLL Hell" problems, before most programmers even knew what DLL Hell was (VersionStamper).

These are tools and components that make important tasks easy or possible – certainly less expensive to do with our tools than from scratch. Yet, one common thread is that we often find ourselves introducing programmers to technologies that they may not be very familiar with. That's why we always say that we at Desaware consider ourselves educators, as well as component vendors.

So far, you've seen the case made for use of state machines in general. If you are experienced with the use of state machines, you already know everything you've read there, and probably more. If you're new to state machines, hopefully you are intrigued enough to realize that they are

worth learning about, regardless of the type of programming you are doing.

In this section, we'll show you how Desaware's StateCoder fits into the picture, and why we not only believe the outrageous claims we're making, but are using it as the infrastructure behind a series of .NET components

What is a StateCoder State Machine?

A state machine in StateCoder is made up of classes that you define that inherit from base classes in the Desaware.StateCoder namespace. The principle is exactly the same as that used to create other .NET classes such as forms, controls and web pages.

The StateMachine object.

In StateCoder a state machine is any object that inherits from the class Desaware.StateCoder.StateMachine or Desaware.StateCoder.UnmanagedStateMachine.

Most of your state machines will derive from Desaware.StateCoder.StateMachine, meaning that it is managed by the StateCoder framework. That means the framework will be responsible for watching for and sending messages, managing threading, synchronization, exceptions and various and other tasks.

The most important part of defining your StateMachine class will be defining the states. This is typically done using attributes. The ContainsState(statename) attribute defines the states for the state machine. You can also pass an array of state objects to the state machine when it is created.

Your StateMachine object is the only object that will be accessible to your main program. You might define properties or methods if you need to specify any information to the state machine before it starts. You might

also add members for any data that needs to be shared by the states in the state machine.

Then you'll call the Start method to start the state machine running.

The StateMachine object has additional capabilities. It can raise events when the state machine enters the end state. It can provide a waitable object for you to wait for completion. It has a hook capability that allows you to monitor state transitions. And, of course, you can build in additional features as well.

But the key thing to remember is that all access to your state machine should be through this object. This object provides much of the encapsulation that helps StateCoder simplify use of state machines (and, of course, does a lot behind the scenes so that you don't have to).

State machines can signal the calling program that they have reached the end state using an event or a wait handle.

The State objects.

Each state in your state machine is represented by a class that inherits from the Desaware.StateCoder.State, Desaware.StateCoder.DynamicState class (the latter allows you to define states dynamically and thus define state machines at runtime), or a generic version of each that accepts the type of the owner state machine as a parameter (VS 2005 only).

One of your state classes must have the "InitialState" attribute, and one the "FinalState" attribute. Each state has a member that points to its state machine, so you can access any data common to the state machine. Each state overrides the method "MessageReceived" to process incoming messages. Each state calls the "NextState" method to specify the next state. It can also override the "EnterState" method that is called when the state is entered (immediately after another state calls NextState). A state object can also process incoming exceptions.

Now here's the cool thing: Once you've created a state machine, you are guaranteed that all messages will come into that state machine on the state machine's thread. You're also guaranteed (obviously) that your state machine will always be in a particular state. This means that unless you go out of your way to create a new thread within the state machine, the only possible area for multithreading synchronization problems are on the public properties of your state machine object. The states themselves are inherently self synchronizing.

But what if you think you do need a new thread inside a state machine? No problem – just create another state machine! StateCoder state machines can use other state machines. This is important, because it means that instead of creating big complex state machines (and their corresponding large complex state diagrams) you'll tend to create smaller state machines that call each other.

The Messages

Desaware's StateCoder defines a message as a .NET object, which means it can be anything you wish based on the needs of the state machine. Messages are generated by objects that implement the IMessageSource interface. This interface includes methods that allow you to determine if a message is ready, to retrieve a message, and to retrieve a wait handle (System.Threading.WaitHandle) that will be signaled when a message is ready.

StateCoder includes a variety of base message source classes, including a generic message source that can wrap any user defined message, and an AsyncResultMessageSource class that works with any .NET class that follows the standard .NET asynchronous call design pattern.

Message sources are a key factor in improving the thread safety of a StateCoder state machine, because while the signaling event for the message may be created on any thread, the message itself will always be

processed on the state machine's own thread – thus eliminating a huge potential source of subtle multithreading problems.

Every state machine can use as many message sources as it needs. Message sources can be defined for the entire state machine, or can be changed for each state.

Another important features is that every StateCoder state machine is itself a message source! A message is considered ready when the state machine enters its end state.

The rest

This is just a quick summary of the features of StateCoder. There's a lot going on behind in the framework to make sure that the state machines use minimal system resources. While you can create state machines to run in their own threads, well designed state machines (especially those that use asynchronous operations) will typically run on the StateCoder thread pool. This is essential for scalability – threads are expensive system resources.

What will you do with StateCoder?

The best way to learn about StateCoder is to download the demo from www.desaware.com. This is a fully functional demo that includes all of the documentation. It's only limitation is that you can only run it with a specific demo assembly. You'll need the full product to use it with your own applications.

We believe that StateCoder will help you create better .NET code more quickly, regardless of whether you are building standalone applications, web services or ASP.NET applications. If you are new to state machines, then you will find the design patterns you learn with StateCoder will become an indispensable tool for all your programming efforts, even when you aren't using the StateCoder framework.

Tutorials

The following five tutorials are designed to help you to become familiar with creating your own state machines. A Quick-Start section for building your own state machine follows this section.

Tutorial 1: Building a state machine for long operations

Let's start with the very simple state machine that consists of performing three long operations consecutively in the form.

```
LongOperation1( )
LongOperation2( )
LongOperation3( )
```

You read earlier about the flaws of the DoEvents approach. You know about the risks and challenges of creating background threads. The StateCoderTutorial1 project demonstrates how you can use StateCoder to not only provide a clean background implementation of this sequence, but to simultaneously perform any number of background sequences simultaneously without any synchronization problems.

The states.vb file contains the state machine implementation. First, you have a class that defines the three long operations. Each one puts a thread to sleep for a specified amount of time.

[VB]
```
Class LongOperations
    Public Shared OperationLength As Integer = 2000
    ' Default 2 seconds
    Public Shared Sub LongOp1()
        Threading.Thread.Sleep(OperationLength)
    End Sub
    Public Shared Sub LongOp2()
        Threading.Thread.Sleep(OperationLength)
    End Sub
```

```
    Public Shared Sub LongOp3()
        Threading.Thread.Sleep(OperationLength)
    End Sub
End Class
```

[C#]

```
class LongOperations
{
   public static int OperationLength = 2000;
      // Default 2 seconds
   public static void LongOp1()
   {
      System.Threading.Thread.Sleep (OperationLength);
   }
   public static void LongOp2()
   {
      System.Threading.Thread.Sleep (OperationLength);
   }
   public static void LongOp3()
   {
   System.Threading.Thread.Sleep (OperationLength);
   }
}
```

The state machine will divide the three long operations into three different states (since in a real scenario, you may wish to perform different operations after each one). Each state begins its long operation when its MessageReceived method is called, and will then set the state to the next state. The following code illustrates this (refer to the VS2005 sample code to see the generic version of this example):

[VB]

```
' The first state runs the first long operation
<InitialState()> Class FirstState
    Inherits State
    Public Overrides Sub MessageReceived(ByVal message _
        As Object, ByVal source As _
        Desaware.StateCoder.IMessageSource)
        LongOperations.LongOp1()
```

```
        nextstate("SecondState")
    End Sub
End Class

Class SecondState
    Inherits State
    Public Overrides Sub MessageReceived(ByVal message _
        As Object, ByVal source As _
        Desaware.StateCoder.IMessageSource)
        LongOperations.LongOp2()
        nextstate("ThirdState")
    End Sub
End Class

Class ThirdState
    Inherits State
    Public Overrides Sub MessageReceived(ByVal message _
        As Object, ByVal source As _
        Desaware.StateCoder.IMessageSource)
        LongOperations.LongOp3()
        nextstate("LastState")
    End Sub
End Class

<FinalState()> Class LastState
    Inherits State
End Class
```

[C#]

```
// The first state runs the first long operation
[InitialState()] class FirstState: State
{
   public override void MessageReceived(object message,
   Desaware.StateCoder.IMessageSource source)
   {
      LongOperations.LongOp1();
      NextState("SecondState");
      }
   }
```

```
class SecondState: State
{
   public override void MessageReceived(object message,
   Desaware.StateCoder.IMessageSource source)
   {
      LongOperations.LongOp2();
      NextState("ThirdState");
   }
}

class ThirdState: State
{
   public override void MessageReceived(object message,
   Desaware.StateCoder.IMessageSource source)
   {
      LongOperations.LongOp3();
      NextState("LastState");
   }
}

[FinalState()] class LastState: State
{
}
```

The final state, as you see, doesn't do anything in this example. You can, if you wish, override the EnterState method of the final state to perform some operation appropriate to the final state of the state machine.

The StateCoderTutorial1 Message Source

In this example we just want to move from one state to the next without delay. This can be accomplished by a message source that is always ready. It doesn't actually have to return a message. The easiest way to accomplish this is to define a message source that always returns True to the MessageReady method as shown here:

[VB]

```
' Here's a message source that is always signaled,
```

```
' and returns a null message
Public Class AlwaysTrueMessageSource
    Inherits GenericMessageSourceBase

    Public Overrides ReadOnly Property MessageReady() _
    As Boolean
        Get
            Return True
        End Get
    End Property

    Public Overrides Function RetrieveMessage() As Object
        Return Nothing
    End Function
End Class
```

[C#]

```
// Here's a message source that is always signaled, and
returns a null message
public class AlwaysTrueMessageSource:
   GenericMessageSourceBase
{
   public override bool MessageReady
   {
      get
      {
         return(true);
      }
   }
   public override object RetrieveMessage()
   {
      return null;
   }
}
```

By inheriting the GenericMessageSourceBase class, all of the work of implementing the wait handle and other default members is handled automatically.

The StateCoderTutorial1 State Machine

Finally comes the state machine class itself. The ContainsState attribute is used to specify the classes for the individual states.

[VB]

```
Imports Desaware.StateCoder
<ContainsState("Desaware.StateCoderTutorial1.FirstState"), _
ContainsState("Desaware.StateCoderTutorial1.SecondState"), _
ContainsState("Desaware.StateCoderTutorial1.ThirdState"), _
ContainsState("Desaware.StateCoderTutorial1.LastState")> _
Public Class ConsecutiveStateMachine
    Inherits StateMachine

    Public Sub New()
        MyBase.New(StateMachineFlags.CreateInNewThread)
        ActiveMessageSource = New AlwaysTrueMessageSource()
    End Sub

    Public SequenceNumber As Integer

End Class
```

[C#]

```
// Attributes to declare the states of our State Machine
[ContainsState("Desaware.StateCoderTutorial1_C.FirstState"),
ContainsState("Desaware.StateCoderTutorial1_C.SecondState"),
ContainsState("Desaware.StateCoderTutorial1_C.ThirdState"),
ContainsState("Desaware.StateCoderTutorial1_C.LastState")]
public class ConsecutiveStateMachine: StateMachine
{
   public ConsecutiveStateMachine():
      base(StateMachineFlags.CreateInNewThread)
   {
      // Our message source will always
            // return a 'signaled'
      ActiveMessageSource = new AlwaysTrueMessageSource();
   }
   public int SequenceNumber;
}
```

Because this state machine is performing very long operations (thus tying up a thread), it doesn't make sense to run the machine on the thread pool (since it would block other state machines on the thread). So the state machine uses the StateMachineFlags.CreateInNewThread flag to tell the framework to run the state machine in its own thread. The message source is set to the AlwaysTrueMessageSource created earlier.

The SequenceNumber is a value to help you keep track of state machines.

That's all it takes.

True, it's a bit more code that long operation calls separated by DoEvents statements, but you'll soon see that the benefits of this approach are even greater than you might expect.

The StateCoderTutorial1 form

The StateCoderTutorial1 form has two buttons and a list control. The form has a variable m_SequenceCounter to keep track of the number of times the sequence (of three long operations) has been run, and identify each one with a number.

```
VB:    Private m_SequenceCounter As Integer
C#:    private int m_SequenceCounter;
```

The RecordCompletion function is called to add a record in the list box when a sequence has completed.

```
[VB]
    Private Sub RecordCompletion(ByVal SequenceNumber As _
        Integer)
        lstSequences.Items.Add("Completed Sequence #: " & _
        SequenceNumber)
    End Sub

[C#]
private void RecordCompletion(int SequenceNumber)
{
```

```
   lstSequences.Items.Add("Completed Sequence #: " +
   SequenceNumber.ToString());
}
```

The cmdNoState_Click method is called when you click the NoState button. This performs the three long operations on the form's main thread. Freezing the form until it completes (as you would expect). This is included just as a reminder of how bad this approach really is. You can add DoEvents statements between the long operations if you wish to see how little it helps and how bad that approach is as well.

```
[VB]
    Private Sub cmdNoState_Click(ByVal sender As _
        System.Object, ByVal e As System.EventArgs) _
        Handles cmdNoState.Click
        LongOperations.LongOp1()
        LongOperations.LongOp2()
        LongOperations.LongOp3()
        RecordCompletion(m_SequenceCounter)
        m_SequenceCounter += 1
    End Sub

[C#]
private void cmdNoState_Click(object sender,
   System.EventArgs e)
{
   LongOperations.LongOp1();
   LongOperations.LongOp2();
   LongOperations.LongOp3();
   RecordCompletion(m_SequenceCounter);
   m_SequenceCounter += 1;
}
```

Starting a state machine is a trivial process as well. You simply create the state machine object and call its Start method. In this example, we're going to wire up the ReachedEndState event of the state machine to the forms ReachedEndState function (which you'll see shortly).

```
[VB]
Private Sub cmdState_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles cmdState.Click
    Dim sm As New ConsecutiveStateMachine()
    sm.SequenceNumber = m_SequenceCounter
    m_SequenceCounter += 1
    AddHandler sm.ReachedEndState, _
    AddressOf ReachedEndState
    sm.Start()
End Sub

[C#]
public delegate void EndStateHandler(object sender);

private void cmdState_Click(object sender,
   System.EventArgs e)
{
   ConsecutiveStateMachine sm = new
   ConsecutiveStateMachine();
   sm.SequenceNumber = m_SequenceCounter;
   m_SequenceCounter += 1;

   EndStateHandler myhandler = new
   EndStateHandler(ReachedEndState);
   sm.ReachedEndState += new
   Desaware.StateCoder.StateMachineBase.
   ReachedEndStateEventHandler(myhandler);

   sm.Start();
}
```

The ReachedEndState method is called by the ReachedEndState event when a state machine completes. Experienced .NET developers know that forms in .NET are not thread safe. This will be discussed further in the next tutorial, but rest assured, with StateCoder, the ReachedEndState event (when attached to any Form or control that derives from Windows.Forms.Control) will be raised in the correct thread.

```
    Private Sub ReachedEndState(ByVal sender As Object)
```

```
        Dim sm As ConsecutiveStateMachine
        sm = CType(sender, ConsecutiveStateMachine)

        RecordCompletion(sm.SequenceNumber)
        ' And dump the state machine
        RemoveHandler sm.ReachedEndState, _
        AddressOf ReachedEndState

        sm.Dispose()

    End Sub


private void ReachedEndState(object sender)
{
   ConsecutiveStateMachine sm;
   sm = (ConsecutiveStateMachine)sender;

   RecordCompletion(sm.SequenceNumber);
   // And dump the state machine

   EndStateHandler myhandler = new
   EndStateHandler(ReachedEndState);
   sm.ReachedEndState -= new
   Desaware.StateCoder.StateMachineBase.
   ReachedEndStateEventHandler(myhandler);
   sm.Dispose();
}
```

The ReachedEndState method also removes the state machine from the event handler, and disposes it.

More on StateCoderTutorial1

There are a couple of other interesting points to note about the operation of this simple state machine.

First of all, you may have noticed that the ConsecutiveStateMachine object is created in a local variable of a function. What happens to the state machine after the function exits?

It's true that the variable to which you assigned the state machine was cleared. However, as soon as you started the state machine, it was taken over by the StateCoder framework. You don't have a reference to it again until it raises its completion event.

Now, it goes without saying that you don't need to take this approach. You can hold references to state machines in your application's variables or in a collection object. If you are going to work with just one instance of a state machine at a time, you might even use the traditional "Dim WithEvents" declaration to allow you to handle the completion event without the AddHandler method call.

However, the approach shown here works just as well, and has the advantage of dramatically simplifying your form's code.

Another interesting consequence of this approach can be seen by clicking multiple times on the "State Machine" button to create multiple state machines. Each time you click on the button, a new state machine is created and passed on to the StateCoder framework, where it is assigned a thread and run.

This really illustrates the power of state machines. A more traditional approach would require you to create your own threads, keep track of them, carefully synchronize any situations where there might be shared information between the threads, and be sure to tear them down on completion.

These StateMachines, by default, have no shared memory, and thus no possibility of synchronization problems. The only place in this example where multithreading is an issue is the ReachedEndState event, each of which is raised in the state machine's thread. However, the only operation that takes place in this event is the Form Invoke method, which marshals the call back to the main form thread, and thus resynchronizes those calls.

You may also be wondering why bother with a message source that is always signaled instead of just calling NextState directly within the EnterState method? You could, arguably, just perform the operations in the EnterState method, using NextState after each operation, instead of bothering with the message source at all.

The reason for doing this is important to understand. Normally all calls into the state objects occur on the state machine thread. The one exception is the very first EnterState call, which takes place on the same thread that calls Start. So if you performed the long operation in the EnterState method, it would tie up the main thread (as would any subsequent EnterState method called through the use of the NextState method). You can read more about this on the section on State machine threading. The fact that the EnterState method of the initial state is called on the thread that calls the Start method does not pose a synchronization problem, because the state machine is not yet running on a StateCoder thread and does not begin to do so until the EnterState method exits. The benefit of this approach is that if you end the state machine (either due to an exception or setting the state to the final state) during the first EnterState method call, the state machine never actually runs, which improves efficiency in that situation.

Tutorial 2: Building a state machine for asynchronous operations.

In the previous example you saw how to implement a state machine that performs a sequence of long operations. Now let's consider a slight variation on the theme in which instead of performing a sequence of long operations, you wish to perform a sequence of asynchronous operations.

Modify the state machine object as follows:

```
[VB]
Imports Desaware.StateCoder
<ContainsState("Desaware.StateCoderTutorial2.FirstState"), _
ContainsState("Desaware.StateCoderTutorial2.SecondState"), _
ContainsState("Desaware.StateCoderTutorial2.ThirdState"), _
ContainsState("Desaware.StateCoderTutorial2.LastState")> _
Public Class ConsecutiveStateMachine
    Inherits StateMachine

    Public Sub New()
        MyBase.New()
    End Sub

    Public SequenceNumber As Integer

End Class

[C#]
// Attributes to declare the states of our State Machine
[ContainsState("Desaware.StateCoderTutorial2_C.FirstState"),
ContainsState("Desaware.StateCoderTutorial2_C.SecondState"),
ContainsState("Desaware.StateCoderTutorial2_C.ThirdState"),
ContainsState("Desaware.StateCoderTutorial2_C.LastState")]
public class ConsecutiveStateMachine: StateMachine
{
   public ConsecutiveStateMachine():base()
   {
   }
```

```
   public int SequenceNumber;
}
```

You'll notice two changes. First, the constructor no longer specifies the StateMachineFlags.CreateInNewThread flag. That's because state machines don't tie up thread resources (they run in a thread pool provided by the .NET framework). So you can allow this state machine to run in the normal StateCoder thread pool.

The other change is that we no longer use the AlwaysTrueMessageSource message source. In asynchronous operations, it's customary for each state (which handles a single asynchronous operation) to set its own message source during the EnterState method.

Before looking at the state objects in detail, let's take a quick look at the asynchronous operations.

```
[VB]
Public Delegate Sub LongOpDelegate()

Class LongOperations
    Public Shared OperationLength As Integer = 2000 '
Default 2 seconds
    Public Shared Sub LongOp1()
        Threading.Thread.Sleep(OperationLength)
    End Sub
    Public Shared Sub LongOp2()
        Threading.Thread.Sleep(OperationLength)
    End Sub
    Public Shared Sub LongOp3()
        Threading.Thread.Sleep(OperationLength)
    End Sub
End Class

[C#]
public delegate void LongOpDelegate();
```

```
class LongOperations
{
   public static int OperationLength = 2000;
   // Default 2 seconds
   public static void LongOp1()
   {
      System.Threading.Thread.Sleep (OperationLength);
   }

   public static void LongOp2()
   {
      System.Threading.Thread.Sleep (OperationLength);
   }

   public static void LongOp3()
   {
      System.Threading.Thread.Sleep (OperationLength);
   }
}
```

At first glance, these look exactly like the long operations performed earlier. The difference here is the presence of a LongOpDelegate type. This delegate makes it possible to invoke the various LongOp methods asynchronously.

The StateTutorial2 State objects

Here's what the Initial state now looks like.

```
[VB]
' The first state runs the first long operation
<InitialState()> Class FirstState
    Inherits State
    Private caller As LongOpDelegate

    Public Overrides Sub EnterState()
        caller = New LongOpDelegate(AddressOf _
        LongOperations.LongOp1)
```

```
        Dim ar As New AsyncResultMessageSource( _
        caller.BeginInvoke( _
        AsyncResultMessageSource.GetAsyncCallbackFunction,_
        Nothing))
        CType(machine, StateMachine).ActiveMessageSource =
        ar
    End Sub

    Public Overrides Sub MessageReceived(ByVal message As _
        Object, ByVal source As _
        Desaware.StateCoder.IMessageSource)
        caller.EndInvoke(CType(message, IAsyncResult))
        nextstate("SecondState")
    End Sub

End Class

[C#]
[InitialState()] class FirstState: State
{

   private LongOpDelegate caller;

   public override void EnterState()
   {
      // Call LongOp1 asynchronously
      caller = new LongOpDelegate(LongOperations.LongOp1);
      AsyncResultMessageSource ar = new
      AsyncResultMessageSource(caller.BeginInvoke(
      AsyncResultMessageSource.GetAsyncCallbackFunction(),
      null));
      ((StateMachine)Machine).ActiveMessageSource = ar;
   }

   public override void MessageReceived(object message,
      Desaware.StateCoder.IMessageSource source)
   {
      // Complete the LongOp1 asynchronous call
      caller.EndInvoke((IAsyncResult)message);
      NextState("SecondState");
   }
```

```
}
```

The state object has a private member named caller. This contains the delegate that will be used to perform the asynchronous message call.

During the EnterState method call, the delegate is initialized to the appropriate LongOp method.

The .NET framework has a common design pattern for virtually all asynchronous operations. The pattern looks like this:

1. Call Beginxxxx to start the operation, returning an IAsyncResult object and specifying an event to call when the operation completes.

1. When the event arrives, call the Endxxx method to complete the operation and retrieve a result.

Rather than create a custom message source for each type of asynchronous operation, the StateCoder framework includes the AsyncResultMessageSource class which can be used with virtually any asynchronous operation.

The constructor for the object takes two parameters. The second is an arbitrary parameter that is not used (or needed) with StateCoder. The first is the result of the Beginxxx method call. In this case, it is the result of the caller delegate's BeginInvoke method call.

BeginInvoke requires you provide a delegate to an event to call when the operation completes. The AsyncResultMessageSource objects simplifies things by allowing you to always pass the result of the AsyncResultMessageSource.GetAsyncCallbackFunction method call for this parameter. This gives you a delegate to a static method to the AsyncResultMessageSource class, which generates the ready signal for the appropriate object when the asynchronous operation completes.

Finally, the AsyncResultMessageSource object is set to be the active message source.

When the operation completes, the MessageReceived method is called with the appropriate IAsyncResult object as the message. This gets passed as a parameter to the caller delegates EndInvoke method.

The other two states work exactly the same way, except they refer to different long operations.

And as for the StateCoderTutorial2 form? It is identical to the previous example.

Asynchronous Operations and Events

Though it may not be obvious, this example illustrates yet another of the services that the StateCoder framework provides for you behind the scenes. You see, in .NET asynchronous operations, the delegate that is called when the operation completes is always called on a thread belonging to the .NET thread pool – which is NOT the same thread as the application's form. As you read earlier, forms in .NET are not thread safe, so any use of the .NET asynchronous operation in a form or UserControl based application must be careful to not only avoid accessing any User interface element, but also to synchronize any access to the form's properties or fields.

The StateCoder framework for managed state machines detects when you are attempting to raise an event to a form, or any object that derives from the Windows.Forms.Control object. It automatically marshals the call to the correct thread for you, eliminating a potential source of serious and hard to detect errors.

Tutorial 3: Building an unmanaged state machine

So far you've seen state machines that represent a simple sequence of states. While you can perform many tasks in this manner, more powerful state machines incorporate decision making and branching within the state machine to control the sequence of states.

In this example, we'll explore the problem of performing a simple transaction on a database. One of the big differences between ADO and ADO.NET is the fact that ADO.NET is connectionless. This poses a number of challenges. Consider the example of inventory management.

In ADO, you might open a connection to a database, read an inventory record, lock it, calculate the new inventory value, and write the updated value into the database. If anyone else attempts to modify the value while you have it locked, they will be blocked until the record is unlocked (or an error will occur).

In ADO.NET, records are not locked. So how do you update an inventory record?

One possibility is to write a smarter update query or use a stored procedure. But in this case we'll show a more generic situation described by the following state machine.

State1: Retrieve the current data value.

State 2: Attempt an update query that uses the original value as a criteria – thus it will only execute if the data value is unchanged.

On success, go to the end state.

On failure, return to State 1 and try again.

State 3: Finish

The UpdateDataMachine state machine

The UpdateDataMachine state machine will have three states, GetCurrentValue, DoTheUpdate and OperationComplete. Unlike previous state machines, this one inherits from the UnmanagedStateMachine class.

Note that in practice you will rarely if ever use unmanaged state machines. But it does have value for helping understand the program flow through a state machine – hence it's inclusion in this tutorial.

The UpdateDataMachine class has three public fields: ItemToModify is the name of the product whose inventory is being updated. In this case it is always "VCR". The ItemsTransacted field is a value indicating the change in inventory you desire, a positive number if you are buying product, negative if you are selling. And the CurrentItemCount field indicates the current inventory value.

```
[VB]
Imports System.Data
Imports Desaware.StateCoder

<ContainsState("StateCoderTutorial3.GetCurrentValue"), _
 ContainsState("StateCoderTutorial3.DoTheUpdate"), _
 ContainsState("StateCoderTutorial3.OperationComplete")> _
 Public Class UpdateDataMachine
    Inherits UnmanagedStateMachine
    Friend conn As OleDb.OleDbConnection
    Friend cmd As OleDb.OleDbCommand

    Public ItemToModify As String
    Public ItemsTransacted As Integer
    Public CurrentItemCount As Integer

    Public Sub New(ByVal dbpath As String)
        conn = New OleDb.OleDbConnection( _
        "Provider=Microsoft.Jet.OLEDB.4.0;Password="""; _
        User " & "ID=Admin;Data Source= """ & dbpath _
        & """;Mode=Share Deny None)
```

```
        conn.Open()
        cmd = New OleDb.OleDbCommand()
        cmd.Connection = conn
    End Sub

End Class

[C#]
// Attributes to declare the states of our State Machine
[ContainsState("StateCoderTutorial3_C.GetCurrentValue"),
ContainsState("StateCoderTutorial3_C.DoTheUpdate"),
ContainsState("StateCoderTutorial3_C.OperationComplete")]
public class UpdateDataMachine:UnmanagedStateMachine
{
   internal System.Data.OleDb.OleDbConnection conn;
   internal System.Data.OleDb.OleDbCommand cmd;

   public string ItemToModify;
   public int ItemsTransacted;
   public int CurrentItemCount;

   public UpdateDataMachine(string dbpath)
   {
      // Intialize a connection to the database
      // string should look like: (any connection string)
      conn = new System.Data.OleDb.OleDbConnection(
      "Provider=Microsoft.Jet.OLEDB.4.0;Password=\"\";
      User ID=Admin;Data Source= \"" + dbpath +
      "\";Mode=Share Deny None;");
      conn.Open();
      cmd = new System.Data.OleDb.OleDbCommand();
      cmd.Connection = conn;
   }
}
```

The GetCurrentValue class has a shadowed Machine property that only serves to perform a type conversion from the base type. All of the classes have the same property in the VS2003 sample code. The VS2005 sample

code avoids the need for this property by inheriting from State(Of UpdateDataMachine) (or state<UpdateDataMachine>). The generics version of the state class handles these type conversions for you. The EnterState method sets the string used to retrieve the value. You may wonder, why set it here and not in the MessageReceived method? The truth is, there is no reason other than to have it there in the event that this state machine were to be adapted later for asynchronous database calls (in which case the query setup and invocation would take place during the EnterState method, and the results processed in the MessageReceived method). But for now, it doesn't matter where the string is set, as long as it is set.

A query to retrieve the inventory value is executed during the MessageReceived class. You might be wondering who calls the MessageReceived class. After all, there is no message source. Hold that thought – you'll see how this happens later.

```
[VB]
<InitialState()> Class GetCurrentValue
    Inherits State
    Private m_ReadString As String

    Protected Shadows ReadOnly Property Machine() _
    As UpdateDataMachine
        Get
            Return (CType(MyBase.Machine, _
            UpdateDataMachine))
        End Get
    End Property

    Public Overrides Sub EnterState()
        m_ReadString = _
        "SELECT Products.Stock FROM(Products)WHERE " & _
        "(((Products.Product)=""" & Machine.ItemToModify _
        & """))"
    End Sub
```

```
    Public Overrides Sub MessageReceived(ByVal message As _
        Object, ByVal source As _
        Desaware.StateCoder.IMessageSource)
        Machine.cmd.CommandText = m_ReadString
        Machine.CurrentItemCount = _
        Machine.cmd.ExecuteScalar()
        NextState("DoTheUpdate")
    End Sub
End Class

[C#]
[InitialState()] class GetCurrentValue: State
{
   private string m_ReadString;

   protected new UpdateDataMachine Machine
   {
      get
      {
         return ((UpdateDataMachine)base.Machine);
      }
   }

   public override void EnterState()
   {
      // Query string that reads the inventory value
      m_ReadString =
      "SELECT Products.Stock FROM(Products)WHERE
      (((Products.Product)=\"" + Machine.ItemToModify +
      "\"))";
   }


   public override void MessageReceived(object message,
      Desaware.StateCoder.IMessageSource source)
   {
      // Retrieve the inventory value
      Machine.cmd.CommandText = m_ReadString;
      Machine.CurrentItemCount =
      (int)Machine.cmd.ExecuteScalar();
      NextState("DoTheUpdate");
```

```
    }
}
```

The DoTheUpdate class attempts to perform the update operation. First, it checks to make sure that you aren't trying to sell something you don't have (checking the newcount variable). If there is insufficient stock, you send an exception to the state machine. This sets the state machine's LastException property to the specified exception and moves the state machine to the end state.

The update query string only works if the current inventory value matches that of the state machine's CurrentItemCount value – meaning the inventory has not been changed since the last time the database was queried.

When the MessageReceived method is called, the update query executes. If it succeeds, the state switches to the OperationComplete state, otherwise it returns to the GetCurrentValue state and tries again.

```
[VB]
Class DoTheUpdate
    Inherits State(Of UpdateDataMachine)
    Private m_UpdateString As String

    Public Overrides Sub EnterState()
        Dim newcount As Integer
        newcount = Machine.CurrentItemCount + _
        Machine.ItemsTransacted
        If newcount < 0 Then
            Machine.SendException(New Exception( _
            "Insufficent stock for purchase"))
            Exit Sub
        End If
        m_UpdateString = _
        "UPDATE Products SET Products.Stock = " & _
        newcount.ToString & _
        " WHERE (((Products.Product)=""" & _
        Machine.ItemToModify & _
```

```
        """) AND ((Products.Stock)=" & _
        Machine.CurrentItemCount.ToString & "))"
    End Sub

    Public Overrides Sub MessageReceived(ByVal message _
        As Object, ByVal source As _
        Desaware.StateCoder.IMessageSource)
        Machine.cmd.CommandText = m_UpdateString
        Dim Lineschanged As Integer
        Lineschanged = Machine.cmd.ExecuteNonQuery()
        If Lineschanged = 0 Then
            Nextstate("GetCurrentValue")
        Else
            NextState("OperationComplete")
        End If
    End Sub
End Class

[C#]
class DoTheUpdate: State<UpdateDataMachine>
{
   private string m_UpdateString;

   public override void EnterState()
   {
      int newcount;
      newcount = Machine.CurrentItemCount +
      Machine.ItemsTransacted;
      if (newcount < 0)
      {
         Machine.SendException(new Exception(
         "Insufficent stock for purchase"));
         return;
      }
      // Query string that updates the inventory value
      // if it is unchanged
      m_UpdateString =
      "UPDATE Products SET Products.Stock = " +
      newcount.ToString() +
      " WHERE (((Products.Product)=\"" +
      Machine.ItemToModify + "\") AND ((Products.Stock)=" +
```

```
      Machine.CurrentItemCount.ToString() + "))";
   }

   public override void MessageReceived(object message,
      Desaware.StateCoder.IMessageSource source)
   {
      Machine.cmd.CommandText = m_UpdateString;
      int Lineschanged;

      Lineschanged = Machine.cmd.ExecuteNonQuery();
      if (Lineschanged == 0)
         // The inventory value was changed by someone
         // else - need to try again
         NextState("GetCurrentValue");
      else
         NextState("OperationComplete");
   }
}
```

The OperationComplete state performs cleanup of the state machine's internal variables, releasing the database connection. It's true, you could perform cleanup by overriding the state machine's Dispose method, but then you're relying on the client to remember to call Dispose. A properly designed state machine[4] always reaches the EnterState method, so it is the appropriate place to clean up internal variables (those that need to be kept around for the client to access after the final state is reached should be cleaned up during the Dispose method).

```
[VB]
<FinalState()> Class OperationComplete
    Inherits State(Of UpdateDataMachine)
```

[4] Managed state machines always execute the EnterState method of the final state. However unmanaged state machines depend on the client to make sure methods are called correctly.

```
    Public Overrides Sub EnterState()
        Machine.cmd.Dispose()
        Machine.conn.Close()
        Machine.conn.Dispose()
    End Sub
End Class

[C#]
[FinalState()] class OperationComplete:
State<UpdateDataMachine>
{

   public override void EnterState()
   {
      // Be sure to clean up
      Machine.cmd.Dispose();
      Machine.conn.Close();
      Machine.conn.Dispose();
   }
}
```

Managing an Unmanaged State Machine

With a regular managed state machine (one that derives from the StateMachine class), the StateCoder framework takes responsibility for a variety of tasks ranging from dispatching messages, managing message sources, to synchronization and so forth. With an unmanaged state machine, you take full responsibility for these tasks.

The sample program performs a simulation, where it executes 100 random transactions, either buying or selling product depending on the state of the option buttons. A list box records the result of each transaction.

A command button toggles the simulation on and off. It does so by changing the state of a timer.

```
[VB]
Private Sub cmdControl_Click(ByVal sender As _
```

```
System.Object, ByVal e As System.EventArgs) _
Handles cmdControl.Click
    If Timer1.Enabled Then
        Timer1.Enabled = False
        cmdControl.Text = "Start"
    Else
        lstStatus.Items.Clear()
        cycles = 0
        Timer1.Enabled = True
        cmdControl.Text = "Stop"
    End If

End Sub

[C#]
private void cmdControl_Click(object sender,
   System.EventArgs e)
{
   if (Timer1.Enabled)
   {
      Timer1.Enabled = false;
      cmdControl.Text = "Start";
   }
   else
   {
      lstStatus.Items.Clear();
      cycles = 0;
      Timer1.Enabled = true;
      cmdControl.Text = "Stop";
   }
}
```

Each timer event performs another simulation cycle, creating the UpdateDataMachine state machine and setting it's fields.

```
[VB]
    Dim dbpath As String = _
    IO.Path.GetFullPath("..\Inventory.mdb")
    Dim m_Random As New Random()
```

```
    Dim cycles As Integer
    Const MAXCYCLES As Integer = 100

    Private Sub Timer1_Elapsed(ByVal sender As _
    System.Object, ByVal e As _
        System.Timers.ElapsedEventArgs) Handles _
        Timer1.Elapsed
        If cycles > MAXCYCLES AndAlso Timer1.Enabled _
        = True Then
            cmdControl_Click(Nothing, Nothing)
        Else
            cycles += 1
        End If
        Dim sm As New UpdateDataMachine(dbpath)
        sm.ItemToModify = "VCR"
        Dim transactioncount As Integer
        If optSelling.Checked Then
            transactioncount = 0 - m_Random.Next(1, 10)
        Else
            transactioncount = m_Random.Next(1, 10)
        End If
        sm.ItemsTransacted = transactioncount
        RunUnmanagedStateMachine(sm)
    End Sub

[C#]
string dbpath = System.IO.Path.GetFullPath(
"..\\..\\Inventory.mdb");
Random m_Random = new Random();
int cycles;
const int MAXCYCLES = 100;

private void Timer1_Elapsed(object sender,
System.Timers.ElapsedEventArgs e)
{
   if ((cycles > MAXCYCLES) && (Timer1.Enabled))
      cmdControl_Click(null, null);
   else
      cycles += 1;

   // Set up the state machine
```

```
   UpdateDataMachine sm = new UpdateDataMachine(dbpath);
   sm.ItemToModify = "VCR";
   int transactioncount;

   if (optSelling.Checked)
      transactioncount = 0 - m_Random.Next(1, 10);
   else
      transactioncount = m_Random.Next(1, 10);

   sm.ItemsTransacted = transactioncount;
   // Run the state machine
   RunUnmanagedStateMachine(sm);
}
```

The RunUnmanagedStateMachine method is called during the timer event to actually run the state machine. It works by calling the sm.SendMessage method repeatedly until the state machine completes. The StateCoder framework does provide some services for you – if a state machine performs a state transition during message processing, the state transition occurs and the state's EnterState method is called. However, all execution of the unmanaged state machine takes place on the thread on which you call the message. And no synchronization is provided should you try to access the state machine on more than one message at a time.

Unlike managed state machines, there is no need for message sources, though you are welcome to use them if you wish. If you do use a message source, it is up to you to wait for a message and dispatch it. The UnmanagedStateMachine class provides no message source services.

```
[VB]
Private Sub RunUnmanagedStateMachine(ByVal sm As _
    UpdateDataMachine)
        Try
            Do While Not sm.MessageReady
                sm.SendMessage(Nothing, Nothing)
            Loop
            If Not sm.LastException Is Nothing Then
                lstStatus.Items.Add( _
```

```
                sm.LastException.Message)
            Else
                If sm.ItemsTransacted > 0 Then
                    lstStatus.Items.Add("Bought: " & _
                    sm.ItemsTransacted.ToString)
                Else
                    lstStatus.Items.Add("Sold: " & CStr( _
                    -sm.ItemsTransacted))
                End If
            End If
        Catch ex As Exception
            lstStatus.Items.Add("Other error: " & _
            ex.Message)
        End Try
        sm.Dispose()
    End Sub

[C#]
private void RunUnmanagedStateMachine(UpdateDataMachine sm)
{
   try
   {
      // Here's a very simple dispatch loop that just _
         keeps sending empty messages
      // to the unmanaged state machine
      while (!sm.MessageReady)
      {
         sm.SendMessage(null, null);
      }

      if (!(sm.LastException == null))
      {
         lstStatus.Items.Add(sm.LastException.Message);
      }
      else
      {
         if (sm.ItemsTransacted > 0)
            lstStatus.Items.Add("Bought: " +
            sm.ItemsTransacted.ToString());
         else
            lstStatus.Items.Add("Sold: " +
```

```
            (-sm.ItemsTransacted).ToString());
      }
   }
   catch (Exception ex)
   {
      lstStatus.Items.Add("Other error: " + ex.Message);
   }
   sm.Dispose();
}
```

When you run this sample program, you will notice that the form remains almost completely frozen during execution (depending on the speed of your system and the duration of the timer, which by default is very short). If you wish the form to be responsive, you'll have to lengthen the timer interval so that it is longer than the typical database operation. This will, of course, slow down the simulation as well.

Tutorial 4: From unmanaged to managed state machines

Tutorial 4 shows the UpdateStateMachine state machine revised to run as a managed state machine. The UpdateStateMachine class is changed to inherit from StateMachine instead of UnmanagedStateMachine.

A managed state machine needs a message source. This poses an interesting dilemma for this example. You could use a message source that is always true (like the AlwaysTrueMessageSource message source defined in tutorial #1. However, in this case we'll use a timer message source with a short duration. This allows us to run the state machine on the thread pool and gives time for other state machines to run if they are present[5]. The AlarmMessageSource message source is included with StateCoder and acts as an alarm or interval timer message source. This line is added to the constructor of the state machine:

```
Me.ActiveMessageSource = New AlarmMessageSource(Nothing, _
                 New TimeSpan(10000))  ' 10ms interval
this.ActiveMessageSource = new AlarmMessageSource(
   TimeSpan.Zero, new TimeSpan(10000));   // 10ms interval
```

The state classes themselves are substantially unchanged.

The real change takes place in the form. While there is no longer a need to run the state machine, you still need to set up the state machine and start it as shown in the new timer event.

---

[5] The StateCoder framework allocates time to every state machine in the thread pool even if its message source is always ready. However, you do have to return from the MessageReceived method in order for other state machines to run. That's why we put each state machine in its own thread in tutorial #1.

```
    Private Sub Timer1_Elapsed(ByVal sender As _
        If Timer1.Enabled = False Then Exit Sub
         ' .NET quirk?
        If cycles > MAXCYCLES Then
            cmdControl_Click(Nothing, Nothing)
        Else
            cycles += 1
        End If
        Dim sm As New UpdateDataMachine(dbpath)
        sm.ItemToModify = "VCR"
        Dim transactioncount As Integer
        If optSelling.Checked Then
            transactioncount = 0 - m_Random.Next(1, 10)
        Else
            transactioncount = m_Random.Next(1, 10)
        End If
        sm.ItemsTransacted = transactioncount
        sm.Start()
        sm.MessageReadySignal.WaitOne()
        ReachedEndState(sm)
        sm.Dispose()
    End Sub
```

```
private void Timer1_Elapsed(object sender,
   System.Timers.ElapsedEventArgs e)
   if (!Timer1.Enabled)
      return;     // .NET quirk?

   if (cycles > MAXCYCLES)
      cmdControl_Click(null, null);
   else
      cycles++;

   // Set up the state machine
   UpdateDataMachine sm = new UpdateDataMachine(dbpath);
   sm.ItemToModify = "VCR";
   int transactioncount;

   if (optSelling.Checked)
      transactioncount = 0 - m_Random.Next(1, 10);
```

```
   else
      transactioncount = m_Random.Next(1, 10);
      sm.ItemsTransacted = transactioncount;
      EndStateHandler myhandler = new
         EndStateHandler(ReachedEndState);
   sm.Start();
   sm.MessageReadySignal.WaitOne();
   ReachedEndState(sm);
   sm.Dispose();
}
```

The state machine's MessageReadySignal property is used to obtain a WaitHandle which is used to wait for the state machine to complete. If it weren't there, you would end up creating potentially hundreds of state machines, which the framework can handle, but will start raising ADO.NET errors as you run out of available connections. The ReachedEndState method processes the message that occurs when the state machine finishes its operation, and records the transactions in the list box.

There is a subtle issue in this example. Why not just add the ReachedEndState event to the event handler and process it that way? The problem is that because the ReachedEndState event is synchronized to the form, it can't be sent while the form thread is in a wait state. The default case has the ReachedEndState event being raised before the state machine's WaitHandle is signaled (this can be modified using the ReachedEndStateEventAfterSignal flag). Even if you were to use this flag to be sure the form will exit the wait state before the event is raised, in this example the next timer event will arrive immediately, placing the thread back in the wait state and thus freezing the application (deadlock). As a general rule you should either respond to the ReachedEndState event, or use the wait handle – but not both.

```
Public Shadows Sub ReachedEndState(ByVal sender _
As Object)
    Dim sm As UpdateDataMachine
```

```
        sm = CType(sender, UpdateDataMachine)
        If Not sm.LastException Is Nothing Then
            lstStatus.Items.Add(sm.LastException.Message)
        Else
            If sm.ItemsTransacted > 0 Then
                lstStatus.Items.Add("Bought: " & _
                sm.ItemsTransacted.ToString)
            Else
                lstStatus.Items.Add("Sold: " & CStr( _
                -sm.ItemsTransacted))
            End If
        End If
    End Sub


public void ReachedEndState(object sender)
{
   UpdateDataMachine sm;
   sm = (UpdateDataMachine)sender;
   if (!(sm.LastException == null))
   {
      lstStatus.Items.Add(sm.LastException.Message);
   }
   else
   {
      if (sm.ItemsTransacted > 0)
         lstStatus.Items.Add("Bought: " +
         sm.ItemsTransacted.ToString());
      else
         lstStatus.Items.Add("Sold: " +
         (-sm.ItemsTransacted).ToString());
   }
}
```

When you try running this project, you'll find that it is really a terrible solution. It is considerably slower than the previous example (mostly because the state machine uses an internal timer that is set to 100 states per second (based on the alarm message source time of 10ms), and partly because of the additional overhead involved in marshaling between

threads). It suffers from the same frozen form as earlier, made worse by the fact that the example actually suspends the user interface thread while waiting for the state machine to complete!

You might wonder why this tutorial was even included. Well, the answer to that will become clear in the next tutorial.

Tutorial 5: Nested state machines

The big problem with the approaches shown in both Tutorial 3 and Tutorial 4 is due to the architecture. We have two different things going on:

6. We have a synchronous long operation – performing the database update.
6. We have a fast loop performing repetitive calls to the synchronous long operation.

In Tutorial 3, we tried running both of these tasks in the main thread, with predictable results (the user interface effectively froze).

One solution might be to convert the inventory update to an asynchronous task. But remember, even though we are using a windows application to demonstrate these state machines, in practice you'll probably be using state machines for other types of applications such as web applications in ASP.NET. In that case, you may well want the operation to complete before you respond to the user, so a synchronous operation is exactly what you want.

So rather than looking for an asynchronous solution to the first task, in Tutorial 5 we fix the application's performance by moving the simulator out of the main application thread. How? Easy – by creating another state machine!

The SimulationMachine state machine contains only two states. The first state runs the UpdateDataMachine state machine. The second state is the final state.

The SimulationMachine class demonstrates several new techniques. First, it has a field named Updater that holds a reference to the UpdateDataMachine state machine. It has a couple of public fields to let the client specify whether the simulator is selling or buying inventory, and

how many cycles remain to complete. The client can terminate the state machine by setting the LeftToDo variable to zero.

We still want the form to receive notification for each transaction. To make this possible, the SimulationMachine has an event to pass notifications to the form, and a method that can be called by the states to cause the event to be raised to the form.

```
[VB]
<ContainsState("StateCoderTutorial5.Inprocess"), _
 ContainsState("StateCoderTutorial5.finished")> _
 Public Class SimulationMachine
    Inherits StateMachine

    Friend DataPath As String
    Friend Updater As UpdateDataMachine

    Public Selling As Boolean

    Public LeftToDo As Integer

    Public Sub New(ByVal dbpath As String)
        DataPath = dbpath
    End Sub

    Public Event TransactionNotificationEvent(ByVal _
    message As String)

    Friend Sub SendTransactionNotification(ByVal message _
    As String)
        RaiseEvent TransactionNotificationEvent(message)
    End Sub

End Class

[C#]

[ContainsState("StateCoderTutorial5_C.Inprocess"),
ContainsState("StateCoderTutorial5_C.finished")]
```

```
public class SimulationMachine: StateMachine
{

   internal string DataPath;  // Path to database
   internal UpdateDataMachine Updater;
   // Nested state machine
   public bool Selling; // True if selling product
   //(subtract from inventory)
   public int LeftToDo;       // Cycles remaning

   public SimulationMachine(string dbpath)
   {
      DataPath = dbpath;
   }

   // Declare event to raise, need to declare a delegate
   public delegate void TransactionHandler(string message);
   // Event to raise to the client on each transaction.
   // Note this event is NOT automatically marshalled if
   // the client is a form
   public event TransactionHandler
   TransactionNotificationEvent;

   // Called by states to raise transaction notification
   internal void SendTransactionNotification(
   string message)
   {
      TransactionNotificationEvent(message);
   }
}
```

The first class, Inprocess, does most of the work for this state machine. You'll see here many of the elements that used to be in the timer event. There is the m_Random variable that generates random numbers. The StartMachine private method creates the UpdateDataMachine state machine, sets up the various fields, and starts the state machine. This method is called both by the EnterState method, and by at the end of the MessageReceived method if the state machine remains in the current state.

A key part of this method is the setting of the ActiveMessageSource property to the newly created UpdateDataMachine. One of the most important features of the StateCoder framework is that state machines are themselves message sources, where the message is considered ready as soon as the state machine terminates. A state machine may override its RetrieveMessage method to return a message, but in this case (in fact, in most cases), the fact that the state machine has completed is all you really care about.

So what is happening here is that the StartMachine method creates a new UpdateDataMachine, registers it as an active message source, then starts it. When that state machine completes its operation, the MessageReceived method will be called.

```
[VB]
<InitialState()> Public Class Inprocess
    Inherits State

    Private m_Random As New Random()

    Protected Shadows ReadOnly Property Machine()
        As SimulationMachine
        Get
            Return CType(MyBase.Machine, SimulationMachine)
        End Get
    End Property

    Private Sub StartMachine()
        Dim submachine As New
        UpdateDataMachine(Machine.DataPath)
        Machine.Updater = submachine
        Machine.ActiveMessageSource = submachine
        submachine.ItemToModify = "VCR"

        Dim transactioncount As Integer

        If Machine.Selling Then
            transactioncount = 0 - m_Random.Next(1, 10)
```

```
        Else
            transactioncount = m_Random.Next(1, 10)
        End If
        submachine.ItemsTransacted = transactioncount

        submachine.Start()
    End Sub

    Public Overrides Sub EnterState()
        StartMachine()
    End Sub

[C#]

[InitialState()] class Inprocess: State
{
   private Random m_Random = new Random();

   protected new SimulationMachine Machine
   {
      get
      {
         return ((SimulationMachine)base.Machine);
      }
   }

   private void StartMachine()
   {
      // Create the internal state machine
      UpdateDataMachine submachine = new
      UpdateDataMachine(Machine.DataPath);
      Machine.Updater = submachine;
      // The UpdateDataMachine is the message source
      Machine.ActiveMessageSource = submachine;
      submachine.ItemToModify = "VCR";

      int transactioncount;
      // Set up the UpdateDataMachine parameters
      if (Machine.Selling)
         transactioncount = 0 - m_Random.Next(1, 10);
      else
```

```
         transactioncount = m_Random.Next(1, 10);

      submachine.ItemsTransacted = transactioncount;

      submachine.Start();
   }

   public override void EnterState()
   {
      StartMachine();
   }
```

When the message is retrieved, the class processes the results of that operation. Here you'll see code that used to be in the form class, where a specific message is generated for each transaction. The SimulationMachine's SendTransactionNotification method passes this method to the state machine class, which in turn raises an event to the form.

If the LeftToDo field indicates that the state machine should continue with the simulation, the StartMachine method is called to create a new UpdateDataMachine and start the process over. Otherwise, the state machine switches to the finished state, which does absolutely nothing.

```
[VB]
Public Overrides Sub MessageReceived(ByVal message _
    As Object, ByVal source As _
    Desaware.StateCoder.IMessageSource)
    Dim udm As UpdateDataMachine
    udm = Machine.Updater
    If Not udm.LastException Is Nothing Then
        Machine.SendTransactionNotification( _
        udm.LastException.Message)
    Else
        If udm.ItemsTransacted > 0 Then
            Machine.SendTransactionNotification( _
            "Bought: " & udm.ItemsTransacted.ToString)
```

```
        Else
            Machine.SendTransactionNotification( _
            "Sold: " & CStr( -udm.ItemsTransacted))
        End If
    End If
    udm.Dispose()
    If Machine.LeftToDo > 0 Then
        Machine.LeftToDo-=1
        StartMachine()
    Else
        nextstate("finished")
    End If
End Sub

End Class

<FinalState()> Public Class finished
    Inherits State
End Class


[C#]
   public override void MessageReceived(object message,
   Desaware.StateCoder.IMessageSource source)
   {
      UpdateDataMachine udm;
      // UpdateDataMachine finished
      //(it generated the message)
      udm = Machine.Updater;
      // Notify the client of a transaction completion
      if (!(udm.LastException == null))
      {
         Machine.SendTransactionNotification(
         udm.LastException.Message);
      }
      else
      {
         if (udm.ItemsTransacted > 0)
            Machine.SendTransactionNotification(
            "Bought: " + udm.ItemsTransacted.ToString());
         else
```

```
            Machine.SendTransactionNotification(
            "Sold: " + (-udm.ItemsTransacted).ToString());
      }
      udm.Dispose();    // Dispose the current machine
      if (Machine.LeftToDo > 0)
      {
         Machine.LeftToDo-=1;
         StartMachine();
      }
      else
         NextState("finished");
   }
}

[FinalState()] public class finished: State
{
}
```

The form is much simpler than in previous examples. The command button event starts the simulation state machine as you've seen earlier. It stops the simulation state machine by setting the LeftToDo field to zero and allowing it to terminate naturally (which is much cleaner than calling the AbortStateMachine method).

```
[VB]
Dim dbpath As String = IO.Path.GetFullPath( _
"..\Inventory.mdb")
Const MAXCYCLES As Integer = 100

Private WithEvents m_SimulationMachine As SimulationMachine

Private Sub cmdControl_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles cmdControl.Click
    If cmdControl.Text = "Start" Then
        lstStatus.Items.Clear()
        m_SimulationMachine = New SimulationMachine(dbpath)
        cmdControl.Text = "Stop"
```

```
        m_SimulationMachine.LeftToDo = MAXCYCLES
        If optSelling.Checked Then _
            m_SimulationMachine.Selling = True
        m_SimulationMachine.Start()
    Else
        cmdControl.Text = "Start"
        m_SimulationMachine.LeftToDo = 0
    End If
End Sub

[C#]

string dbpath = System.IO.Path.GetFullPath(
"..\\..\\Inventory.mdb");
const int MAXCYCLES = 100;

private SimulationMachine m_SimulationMachine;

private void cmdControl_Click(object sender,
System.EventArgs e)
{
   if (cmdControl.Text == "Start")
   {
      lstStatus.Items.Clear();
      m_SimulationMachine = new SimulationMachine(dbpath);
      m_SimulationMachine.TransactionNotificationEvent
      += new SimulationMachine.TransactionHandler(
      SimulationMachine_TransactionNotificationEvent);
      m_SimulationMachine.ReachedEndState+= new
      Desaware.StateCoder.StateMachineBase.
      ReachedEndStateEventHandler(
      m_SimulationMachine_ReachedEndState);
      cmdControl.Text = "Stop";
      m_SimulationMachine.LeftToDo = MAXCYCLES;
      if (optSelling.Checked)
         m_SimulationMachine.Selling = true;
      m_SimulationMachine.Start();
   }
   else
   {
      cmdControl.Text = "Start";
```

```
      m_SimulationMachine.LeftToDo = 0;
   }
}
```

The SimulationMachine's TransactionNotificationEvent requires special handling. While the ReachedEndState event of a state machine automatically marshals to the correct thread for a form, automatic synchronization is not provided for other events that you define in a state machine. The following code shows how you should invoke a method on the form using a delegate by calling the form's Invoke method to marshal the call to the correct thread.

```
[VB]
Private Delegate Sub SetListStringFunc(ByVal _
message As String)

Private Sub SetListString(ByVal message As String)
    lstStatus.Items.Add(message)
End Sub

Private Sub _
    m_SimulationMachine_TransactionNotificationEvent ( _
        ByVal message As String) Handles _
        m_SimulationMachine.TransactionNotificationEvent
        Me.Invoke(New SetListStringFunc(AddressOf _
        SetListString), New String() {message})
    End Sub

[C#]

private delegate void SetListStringFunc(string message);

private void SetListString(string message)
{
   lstStatus.Items.Add(message);
}
```

```
private void
m_SimulationMachine_TransactionNotificationEvent(
string message)
{
   SetListStringFunc mySetListStringHandler =
   new SetListStringFunc(SetListString);
   this.Invoke(mySetListStringHandler, new
   string[]{message});
}
```

The ReachedEndState event in this example is only used for cleanup. We don't really care when the simulation state machine actually finishes.

```
[VB]
Private Sub m_SimulationMachine_ReachedEndState(ByVal _
    sender As Object) Handles _
    m_SimulationMachine.ReachedEndState
    m_SimulationMachine.Dispose()
    cmdControl.Text = "Start"
    m_SimulationMachine = Nothing
End Sub

[C#]

private void m_SimulationMachine_ReachedEndState(
   object sender)
{
      m_SimulationMachine.TransactionNotificationEvent-=
      new SimulationMachine.TransactionHandler(
      m_SimulationMachine_TransactionNotificationEvent);
      m_SimulationMachine.ReachedEndState-= new
      Desaware.StateCoder.StateMachineBase.
      ReachedEndStateEventHandler(
      m_SimulationMachine_ReachedEndState);
      cmdControl.Text = "Start";
      m_SimulationMachine.Dispose();
      m_SimulationMachine = null;
}
```

You'll find that this example works much better than the previous two. Not only does the form remain responsive, but you'll see the list box being updated in real time as the inventory is updated.

Review

This concludes the introductory tutorials. You've learned the basics of creating state machines using StateCoder. You've seen how they can be applied both to handle long background operations and asynchronous operations. You haven't seen large numbers of synchronization objects and SyncLock calls – because the StateCoder framework is largely self-synchronizing. You've seen how to create a custom messages source, how to use several of the built in message sources, and how to let state machines call other state machines and use them as message sources.

Now it's time to look at some more realistic examples of using StateCoder.

StateCoder Quick Start

Creating a Simple Managed State Machine

There are 9 steps involved when creating a simple managed state machine using StateCoder. This section will summarize each step then guide you through these steps to create a simple timer-based state machine.

Step 1 – Create Project

Create a new .NET Windows Application project (you can create other project types but for the sake of simplicity, this quick start uses a Windows Application project), add a new "States" class file for the state machine (you may also add a copy of the "States" template file). Add a reference from your project to the .NET assembly "Desaware StateCoder".

Step 2 – Assign Names

After designing your state machine, assign names to your state machine including namespace, state machine name, and names for each state.

Step 3 – Editing your "States" class file

Add the "`Imports Desaware.StateCoder`" (VB) or "`using Desaware.StateCoder;`" (C#) statements to the top of the "States" class file. If you are using VB, we also recommend that you add the "`Option Explicit On`" and "`Option Strict On`" code statements to the top of your States class file or make sure these are set in your project settings.

Step 4 – Create your StateMachine class

Declare your state machine class object and inherit it from the "StateMachine" class object. You will generally declare your state machine object as a public object. Assign the "`ContainsState`" attribute to your state machine for each state it supports.

[VB]

```
<ContainsState("YourStateMachineNameSpace.FirstState"), _
ContainsState("YourStateMachineNameSpace.SecondState"), _
ContainsState("YourStateMachineNameSpace.ThirdState"), _
ContainsState("YourStateMachineNameSpace.LastState")> _
Public Class YourStateMachine
    Inherits StateMachine
```

[C#]

```
[ContainsState("YourStateMachineNameSpace.FirstState"),
ContainsState("YourStateMachineNameSpace.SecondState"),
ContainsState("YourStateMachineNameSpace.ThirdState"),
ContainsState("YourStateMachineNameSpace.LastState")]
public class YourStateMachine: StateMachine
```

Step 5 – Create your State classes

Add your State class objects and inherit them from the "State" class object or it's generic version. You will generally declare your state classes as a `Friend` (VB) or `internal` (C#) object (assuming that the state machine and state classes are in the same file). Assign the "`InitialState`" and "`FinalState`" attributes to your first and last state classes.

[VB]

```
<InitialState()> Friend Class FirstState
    Inherits State
or
    Inherits State(Of YourStateMachine)
   :
   :
<FinalState()> Friend Class LastState
    Inherits State
or
    Inherits State(Of YourStateMachine)
```

[C#]

```
[InitialState()] internal class FirstState: State
or
[InitialState()] internal class FirstState:
State<YourStateMachine>

   :

   :
[FinalState()] internal class LastState: State
or
[FinalState()] internal class LastState:
State<YourStateMachine>
```

Step 6 – Add functions to your state machine

Generally, you want to include one or more constructors for your state machine class. In your state machine constructor, you would usually assign the state machine's `StateMachineFlags` (if you don't want to use the default ThreadPool) by calling the base class's constructor, and set the state machine's `ActiveMessageSource` or `ActiveMessageSources` property to assign a message source. If you do

not assign a message source in the state machine's constructor, be sure to assign one before you exit the first state's `EnterState` subroutine, otherwise an exception will be raised.

[VB]

```
Public Sub New()
    MyBase.New(StateMachineFlags.CreateInNewThread Or _
    StateMachineFlags.ForceEndStateOnAbort)
    ' We assign a 5 second timer message source
    ActiveMessageSource = New _
    Desaware.StateCoder.AlarmMessageSource( _
    TimeSpan.FromSeconds(5), TimeSpan.FromSeconds(5))
End Sub
```

[C#]

```
public
MyStateMachine():base(StateMachineFlags.CreateInNewThread |
StateMachineFlags.ForceEndStateOnAbort)
{
    // We assign a 5 second timer message source
    this.ActiveMessageSource = new
    Desaware.StateCoder.AlarmMessageSource(
    TimeSpan.FromSeconds(5), TimeSpan.FromSeconds(5));
}
```

Step 7 – Add functions to your states

Generally, your state classes (with the exception of the last state) will include an override of the `MessageReceived` subroutine where you would normally attach code to transition to the next state. Your state classes may also include an override of the `EnterState` subroutine where you would normally attach code to start this particular state. If you

need to access the state machine from any of your state classes and you're not using the generic version of the state class, you'll probably want to shadow the state machine's readonly `Machine` property to retrieve a reference to the state machine object. Another common function to override is the state object's `ExceptionReceived` subroutine. Override this subroutine to do your own exception handling or to prevent the state machine from automatically ending.

[VB]

```
<InitialState()> Friend Class FirstState
    Inherits State

   ' The Machine property isn't used if inheriting from
   ' the generic State(Of …) class.
    Protected Shadows ReadOnly Property Machine() _
    As MyStateMachine
        Get
            Return CType(MyBase.Machine, MyStateMachine)
        End Get
    End Property

    Public Overrides Sub EnterState()
        ' Run your code for this state
    End Sub

    Public Overrides Sub MessageReceived(ByVal message _
    As Object, ByVal source As _
    Desaware.StateCoder.IMessageSource)
        ' transition to the next state
        NextState("SecondState")
    End Sub
End Class
```

[C#]

```
[InitialState()] internal class FirstState: State
{

   // The Machine property isn't used if inheriting from
   // the generic State(Of …) class.
     protected new MyStateMachine Machine
     {
           get
           {
                return ((MyStateMachine)base.Machine);
           }
     }

     public override void EnterState()
     {
           // Run your code for this state
     }

     public override void MessageReceived(object message,
     Desaware.StateCoder.IMessageSource source)
     {
          // transition to the next state
          NextState("SecondState");
     }
}
```

Step 8 – Start your State Machine

Create an instance of your state machine object, then call the `Start` function to start your state machine. StateCoder will call the `EnterState` function of the first state before returning from the `Start` function call. You can assign a delegate to the state machine's `StateTransitionMonitor` property. The `StateTransitionMonitor` property is used to monitor state transitions and will call your delegate each time a state transition is about to occur.

This feature is strictly used to monitor state transitions, do not use it to alter state transitions. You can also assign a delegate to the state machine's `ReachedEndState` event. The `ReachedEndState` event is use to notify you that the state machine has completed.

[VB]

```
mysm = New MyStateMachine()
' Hook the state transition event to our function
mysm.StateTransitionMonitor = AddressOf _
StateTransitionEvent
' Hook the end state event to our function
AddHandler mysm.ReachedEndState, AddressOf _
ReachedEndStateEvent
' Start the state machine
mysm.Start()

Private Sub StateTransitionEvent(ByVal EnteringState _
    As String)
    ' State transitioned to "EnteringState"
End Sub

Private Sub ReachedEndStateEvent(ByVal sender As Object)
    ' State machine completed
End Sub
```

[C#]

```
mysm = new MyStateMachine();
// Hook the state transition event to our function
mysm.StateTransitionMonitor = new
Desaware.StateCoder.StateTransition
(MyStateTransitionEvent);
// Hook the end state event to our function
EndStateHandler myhandler = new
EndStateHandler(MyReachedEndStateEvent);
```

```
mysm.ReachedEndState += new
Desaware.StateCoder.StateMachineBase.ReachedEndStateEventHa
ndler(myhandler);
// Start the state machine
mysm.Start();

private delegate void EndStateHandler(object sender);

private void MyReachedEndStateEvent(object sender)
{
     // State machine completed
}

private void MyStateTransitionEvent(string enteringstate)
{
     // State transitioned to "enteringstate"
}
```

Step 9 – Wait for your State Machine to end

Attach code to your ReachedEndStateEvent function. This will be called by StateCoder if you had earlier set your state machine's ReachedEndState event to a delegate handler. You should remove the delegate handler from your state machine object, and dispose of your state machine in this function.

[VB]

```
Private Sub ReachedEndStateEvent(ByVal sender As Object)
    Dim mysm As MyStateMachine
    mysm = CType(sender, MyStateMachine)
    ' State machine completed, do cleanup
    RemoveHandler mysm.ReachedEndState, AddressOf _
    ReachedEndStateEvent
    mysm.Dispose()
End Sub
```

[C#]

```
private void MyReachedEndStateEvent(object sender)
{
     MyStateMachine mysm;
     mysm = (MyStateMachine)sender;
     // State machine completed, do cleanup
     EndStateHandler myhandler = new
     EndStateHandler(MyReachedEndStateEvent);
     mysm.ReachedEndState -= new
Desaware.StateCoder.StateMachineBase.ReachedEndStateEventHa
ndler(myhandler);
     mysm.Dispose();
}
```

Example: Creating a simple timer based state machine

This sample will go through the steps in creating a simple state machine that uses a timer to transition to the next state. The project source is located in the Quick Start folder.

Step 1 – Create Project

Create a new .NET Windows Application project named QuickStart (both assembly and namespace). Add a new “States” class file to this project (you may instead include a copy of the “States” template file to this project). Add a reference from your project to the .NET assembly “Desaware StateCoder”.

Step 2 – Assign Names

The state machine will be named QuickStartStateMachine. This state machine will contain four states named FirstState, SecondState,

ThirdState, and FinalState. The FirstState will have the InitialState attribute and the LastState will have the FinalState attribute.

Step 3 – Editing your "States" class file

Add the "`Imports Desaware.StateCoder`" (VB) or "`using Desaware.StateCoder;`" (C#) statements to the top of the "States" class file. If you are using VB, we also recommend that you add the "`Option Explicit On`" and "`Option Strict On`" code statements to the top of your States class file.

Step 4 – Create your StateMachine class

Declare the QuickStartStateMachine state machine class object and inherit it from the "StateMachine" class object.

[VB]

```
<ContainsState("QuickStart.FirstState"), _
ContainsState("QuickStart.SecondState"), _
ContainsState("QuickStart.ThirdState"), _
ContainsState("QuickStart.LastState")> _
Public Class QuickStartStateMachine
    Inherits StateMachine
```

[C#]

```
namespace QuickStart
{
   // Attributes to declare the states of your State
Machine
   [ContainsState("QuickStart.FirstState"),
   ContainsState("QuickStart.SecondState"),
   ContainsState("QuickStart.ThirdState"),
   ContainsState("QuickStart.LastState")]
```

```
public class QuickStartStateMachine: StateMachine
{
```

Step 5 – Create your State class

Create the FirstState, SecondState, ThirdState, and LastState class objects, inherit them from the "State" class object. Declare the state classes as a `Friend` (VB) or `internal` (C#) object. Assign the "`InitialState`" and "`FinalState`" attributes to the FirstState and LastState classes. See the example code for the generics version of this example.

[VB]

```
<InitialState()> Friend Class FirstState
    Inherits State
End Class

Friend Class SecondState
    Inherits State
End Class

Friend Class ThirdState
    Inherits State
End Class

<FinalState()> Friend Class LastState
    Inherits State
End Class
```

[C#]

```
[InitialState()] internal class FirstState: State
{
}
```

```
internal class SecondState: State
{
}

internal class ThirdState: State
{
}

[FinalState()] internal class LastState: State
{
}
```

Step 6 – Add functions to your state machine

Add a constructor to the state machine. The constructor will call the StateMachine base class's constructor passing it the CreateInNewThread and ForceEndStateOnAbort flags. The constructor will call the StateMachine base class's SetTraceLevel to display tracing information when the project runs in the Visual Studio environment. Finally, the constructor sets the state machine's ActiveMessageSource property to a new AlarmMessageSource object which sets the message source to a 5 second interval alarm.

[VB]

```
Public Sub New()
    MyBase.New(StateMachineFlags.CreateInNewThread Or _
    StateMachineFlags.ForceEndStateOnAbort)
    MyBase.SetTraceLevel(SCTraceSwitch.TraceOptions.All)
    ActiveMessageSource = New _
    Desaware.StateCoder.AlarmMessageSource( _
    TimeSpan.FromSeconds(5), TimeSpan.FromSeconds(5))
End Sub
```

C#

```
public
QuickStartStateMachine():base(StateMachineFlags.CreateInNew
Thread | StateMachineFlags.ForceEndStateOnAbort )
{
     QuickStartStateMachine.SetTraceLevel
     (SCTraceSwitch.TraceOptions.All);
      this.ActiveMessageSource = new
     Desaware.StateCoder.AlarmMessageSource(
     TimeSpan.FromSeconds(5), TimeSpan.FromSeconds(5));
}
```

Step 7 – Add functions to your states

For our sample, we will just override the `MessageReceived` subroutine where we add code to transition to the next state.

[VB]

```
' In the FirstState Class
Public Overrides Sub MessageReceived(ByVal message As _
Object, ByVal source As Desaware.StateCoder.IMessageSource)
    NextState("SecondState")
End Sub

' In the SecondState Class
Public Overrides Sub MessageReceived(ByVal message As _
Object, ByVal source As Desaware.StateCoder.IMessageSource)
    NextState("ThirdState")
End Sub

' In the ThirdState Class
Public Overrides Sub MessageReceived(ByVal message As _
Object, ByVal source As Desaware.StateCoder.IMessageSource)
    NextState("LastState")
End Sub
```

[C#]

```
// In the FirstState Class
public override void MessageReceived(object message,
Desaware.StateCoder.IMessageSource source)
{
     NextState("SecondState");
}

// In the SecondState Class
public override void MessageReceived(object message,
Desaware.StateCoder.IMessageSource source)
{
     NextState("ThirdState");
}

// In the ThirdState Class
public override void MessageReceived(object message,
Desaware.StateCoder.IMessageSource source)
{
     NextState("LastState");
}
```

Step 8 – Start your State Machine

Add a label and command button to your form and attach the following code to the button's click event. For our sample state machine, we want to be notified when a state changes, and also when the state machine ends. We assign a delegate to the state machine's StateTransitionMonitor property to inform us when a state is about to change. We hook a handler to the state machine's ReachedEndState event to notify us when the state machine ends.

[VB]

```
Dim qs_sm As QuickStartStateMachine

qs_sm = New QuickStartStateMachine()
' Hook the state transition event to our function
qs_sm.StateTransitionMonitor = AddressOf _
StateTransitionEvent
' Hook the end state event to our function
AddHandler qs_sm.ReachedEndState, AddressOf _
ReachedEndStateEvent
' Start the state machine
qs_sm.Start()
```

[C#]

```
QuickStartStateMachine qs_sm;
qs_sm = new QuickStartStateMachine();

// Hook the state transition event to our function
qs_sm.StateTransitionMonitor = new
 Desaware.StateCoder.StateTransition
(StateTransitionEvent);

// Hook the end state event to our function
EndStateHandler myhandler = new
EndStateHandler(ReachedEndStateEvent);
qs_sm.ReachedEndState += new
Desaware.StateCoder.StateMachineBase.ReachedEndStateEventHa
ndler(myhandler);

// Start the state machine
qs_sm.Start();
```

Finally, declare the delegate functions in your Form class.

[VB]

```
Private Sub StateTransitionEvent(ByVal EnteringState _
As String)
    Label1.Text = "State transitioned to " + EnteringState
End Sub

Private Sub ReachedEndStateEvent(ByVal sender As Object)
End Sub
```

[C#]

```
private delegate void EndStateHandler(object sender);

private void StateTransitionEvent(string enteringstate)
{
   Label1.Text = "State transitioned to " + enteringstate;
}

private void ReachedEndStateEvent(object sender)
{
}
```

Step 9 – Wait for your State Machine to end

Attach code to the ReachedEndStateEvent function for cleanup. Remove the delegate handler from the state machine object and dispose the state machine in this function.

[VB]

```
Private Sub ReachedEndStateEvent(ByVal sender As Object)
    Dim qssm As QuickStartStateMachine

    qssm = CType(sender, QuickStartStateMachine)
```

```
    RemoveHandler qssm.ReachedEndState, AddressOf _
    ReachedEndStateEvent
    Label1.Text = "State machine completed"
    qssm.Dispose()
End Sub
```

[C#]

```
private void ReachedEndStateEvent(object sender)
{
   QuickStartStateMachine qs_sm;
   qs_sm = (QuickStartStateMachine)sender;

   Label1.Text = "State machine completed";

   // And dump the state machine
   EndStateHandler myhandler = new
EndStateHandler(ReachedEndStateEvent);
   qs_sm.ReachedEndState -= new
Desaware.StateCoder.StateMachineBase.ReachedEndStateEventHa
ndler(myhandler);
   qs_sm.Dispose();
}
```

Run the State Machine and watch the state changes updated on the label control. You can start more than one state machine by repeatedly clicking on the command button. Look at the Output window in Visual Studio while the State Machine is running and look at the tracing information output from the State Machine SetTraceLevel property. One question you may have is why is the State Machine variable declared as a local variable in the command button's click event. Can that variable be garbage collected after the State Machine starts? The answer is that once you start the State Machine, StateCoder holds a reference to your State Machine object so it will not be garbage collected until after the state machine ends.

Sample Applications

State machines are a fundamental concept in programming. It is therefore impossible for us to even begin to list all the places where you might end up using StateCoder. We've developed a number of examples that we think illustrate the capabilities of the package, and help you see the range of possibilities. In this section you'll find a brief description of each sample application and some insight into its architecture. Implementation details for each application can be found by reading through the application code itself.

Retrieving Information from the Internet

You saw in the tutorial section how a state machine can be deployed to perform a series of asynchronous operations. One of the most common asynchronous operations you'll see is that of requesting information through networks, especially calls to read data from web sites or execute web service calls.

The AmazonRank application solves a problem that we were interested in. We wanted the ability to quickly read a list of book rankings from Amazon.com (our president, Dan Appleman, has five books and four e-books in print, and we have an office pool every week to guess the average ranking[6]).

This application demonstrates the following features:

- Using a state machine to perform a series of asynchronous operations.
- Implementing web requests in a state machine.

[6] Just kidding. Dan made us say that. He really wanted it so he could monitor not only his books, but others that he works on in his additional role as editorial director at Apress.

- Extracting data from a web page using Regular Expressions[7].
- Using isolated storage.
- Marshaling state machine events to form threads.

[7] Refer to Dan Appleman's EBook "Regular Expressions with .NET" for details on using Regular Expressions.

Do it yourself transactioning

This application (found in the Samples\Transactioning directory of your StateCoder installation) demonstrates the following features:

- Implementing transactioning using StateCoder
- Building your own inheritable state machines
- Sharing state objects among state machines
- Tracing
- Nesting of state machines.
- Testing and simulation of state machines

About Transactioning

Transactioning is a subject that, for all that has been written about it, can be rather confusing – especially when programming for Windows. The problem is that the terminology and subsystems that handle transactions keep changing.

Here then, is a one page explanation of transactioning that will hopefully put everything into context, and give us a common ground in order to discuss transactioning in the context of StateCoder and state machines.

What is a transaction?

It is a sequence of operations that meet the following requirements:

A: The entire sequence of operations occurs. If any one fails, then any effects of any of the operations must be reversed so that it is as if none of them had occurred (**A**tomicity)

C: The data managed by the transaction must remain **C**onsistent. It may not be corrupted or made invalid by the transaction, whether it succeeds or fails.

I: Each transaction is **I**solated from all others. It runs as if it is the only one in the system.

D: The results of a transaction are **D**urable: Once executed successfully, the results are stored (on disk or other reasonably permanent location).

Certainly a transaction server such as those in Enterprise Services is suitable for many applications, especially complex transactions and enterprise solutions. However, any algorithm that meets the ACID requirements meets the definition of a transaction. In many cases it makes sense to rely on other tools to implement transactions – for example: in a database backed web application you may find that the transactioning provided by your database is more than sufficient for your purposes.

It turns out that state machines are great tools for implementing transactions. After all:

- A state machine defines a series of operations.
- You have complete control over the order of operations, and knowledge of whether each one has succeeded or failed.
- StateCoder based state machines are inherently independent of each other. The members of each state machine class and its individual state objects cannot be accessed by any other[8].
- Consistency and Durability can be incorporated into your design. For example: if you save your results during the final state, you can be certain that your state machine will always save its results[9].

The StateCoderAutoBid and StateCoderAuctionDatabase (in the Transactions directory) demonstrate a simple transactioning scheme. This example takes advantage of the rather brute force record locking scheme

---

[8] Of course, you could figure out a way to do this, but it would take some extra effort on your part.

[9] Barring an invalid application termination, of course.

of opening an XML database for exclusive access – not something you would want to do in real life (you'd be much more likely to use update queries such as shown in tutorials 3 through 5), but it does illustrate the point nicely.

The sample application divides into the following parts:

The StateCoderAuctionDatabase component.

This component manages all access to the underlying AuctionInfo.xml database, a simple auction database that contains a list of registered users, a list of items being auctioned, and the current price and bidder.

It implements a number of state machines:

- The adbsm state machine: This is a base state machine object that provides functions to perform operations on the XML database. It is designed to be inherited by other state machines that perform specific operations.
- The UserListSM state machine: Derives from adbsm. This state machine retrieves a list of users from the XML database.
- The AuctionListSM state machine: Derives from adbsm. This state machine retrieves from the XML database a list of items available for auction.
- The AuctionInfoSM state machine: Derives from adbsm. This state machine retrieves from the XML database information about a single item being auctioned.
- The BidOnItemSM state machine: Derives from adbsm. This state machine enters a bid transaction on the database. The bid either succeeds or fails (atomicity). The state machine insures that either all or none of the fields for an item are updated (consistency). The state machine guarantees that it cannot be interfered by other simultaneous bids (locks out other access until it is finished –

isolation). And the state machine makes sure the bid is stored in the database if it succeeds, or the contents of the database remain unchanged if it fails (durability).

- The AuctionBidSM state machine: Derives from adbsm. Performs automatic bidding on an item starting with the current price, applying an increment if a bid fails, and continuing until either the maximum price is reached or the bid succeeds.

Because many of the state machines perform similar tasks, they can actually share the same state classes.

The StateCoderAutoBid Application

As much fun as it is to experiment with a single state machine, the real fun doesn't begin until you start experimenting with larger numbers of state machines. The AutoBid application tests the StateCoderAuctionDatabase component by creating a large number of bidders, each of which is trying to purchase an item at the lowest price possible, up to a specified maximum price. This allows you to stress test the state machine and make sure that it is, in fact, implementing the transactions correctly.

Building Dynamic State Machines

This application demonstrates:

- Dynamic creation of a state machine based on an XML database.
- Various late bound dynamic invocation techniques.
- Marshaling state machine to form threads.
- Using the QueuedStream stream class.
- Using the ParsingStreamReader message source.

About Dynamic State Machines

So far all of the state machines you have seen have been defined purely in code. The states are defined statically using the ContainsState attribute. The StateCoder framework is responsible for actually creating instances of your state objects.

The StateCoder framework also makes it possible to define state machines dynamically “on the fly.” This is accomplished by creating the state objects in your application and passing them as an array to your state machine class. When creating dynamic state machines, you need not use the <InitialState> and <FinalState> attributes to identify the start and end state. The StateCoder framework assumes that the first state in the array is the initial state and the last state in the array is the final state. As with static state machines, you must have two states in each state machine.

It is important to distinguish between unmanaged/managed state machines and static/dynamic states. Every combination of state machine and state is allowed as shown in the following table:

State Machine \ State	Static	Dynamic
Unmanaged	States defined using the ContainsState attribute. Messages dispatched by your code.	States created by your code. Messages dispatched by your code.
Managed	States defined using the ContainsState attribute. States run on the StateCoder thread pool and messages are dispatched by the StateCoder framework.	States created by your code. States run on the StateCoder thread pool and messages are dispatched by the StateCoder framework.

State objects in a dynamic state machines always derive from the DynamicState class.

The CommandLine Sample Application

The CommandLine sample application implements a simple tool that allows you to invoke shared methods and properties of the .NET framework directly from the command line. However, rather than making you find the method and enter the full namespace name for the method, this application divides the supported methods into categories and provides simple names that map into the various commands.

Each category represents an operating “mode” that you enter using a command, and exit using another command. Thus, by typing “Diagnostics” you enter diagnostic mode where certain commands are supported. You can then type “Exit” to exit diagnostic mode and either enter another mode, or exit the application by typing “Exit” again.

Each “mode” represents a different state in the state machine. The states are generated at runtime based on the contents of the XML file.

The following XML is an example of the input to the CommandLine program:

```
<?xml version="1.0" encoding="utf-8"?>
<StateMachineSchema
xmlns="http://tempuri.org/XMLStates.xsd">
    <State>
        <Name>InitialState</Name>
        <Input>
            <Pattern>Diagnostics</Pattern>
            <NextState>DiagnosticState</NextState>
        </Input>
        <Input>
            <Pattern>Application</Pattern>
            <NextState>ApplicationState</NextState>
        </Input>
        <Input>
            <Pattern>Exit</Pattern>
            <NextState>FinalState</NextState>
        </Input>
    </State>
    <State>
        <Name>DiagnosticState</Name>
        <Input>
            <Pattern>Processes</Pattern>

<Execute>System.Diagnostics.Process.GetProcesses</Execute>
        </Input>
        <Input>
            <Pattern>Tracing</Pattern>

<Execute>System.Diagnostics.Trace.Listeners</Execute>
        </Input>
        <Input>
            <Pattern>Exit</Pattern>
            <NextState>InitialState</NextState>
        </Input>
```

```
    </State>
    <State>
        <Name>ApplicationState</Name>
        <Input>
            <Pattern>Domain</Pattern>

<Execute>System.AppDomain.CurrentDomain</Execute>
        </Input>
        <Input>
            <Pattern>Thread</Pattern>

<Execute>System.AppDomain.GetCurrentThreadId</Execute>
        </Input>
        <Input>
            <Pattern>Exit</Pattern>
            <NextState>InitialState</NextState>
        </Input>
    </State>
    <State>
        <Name>FinalState</Name>
    </State>
</StateMachineSchema>
```

The <State> tag defines a state. Within a <State> tag you have:

- The <Name> tag specifies the name of the state. Each state name must be unique.
- One or more <Input> tags represent a user's command line input into the state and response to that input
  Within the <Input> tag:
    - The <Pattern> tag specifies the actual user input, or "message" into the state for this <Input> block.
    - The <NextState> tag specifies the state to go to after processing this input.
    - The <Execute> tag specifies the method to execute when this pattern is detected.

A schema file is provided to allow verification of the XML before processing of the file.

Refer to the application code for further details regarding the implementation of this state machine.

Improving performance in dynamic ASP.NET web sites

This application demonstrates:

- Use of StateCoder in the context of ASP.NET applications.
- Dynamic graphics in ASP.NET

Predictive ASP.NET and Desaware's StateCoder

If you read Microsoft's documentation and various articles about how to create ASP.NET development, there are one theme that you will hear over and over:

> Always use stateless components because stateless components improve scalability

All of ASP.NET's components and web controls are built based on this principle. We're told that objects should not be stored in the application's session variables. Any information that needs to be stored between requests is stored in hidden fields on the page, as cookies, in a database, or using any of the other persistence techniques supported by ASP.NET. By avoiding the accumulation of objects, this approach results in improved scalability.

But this approach comes with a price. And that price is in the form of performance.

To understand the price, it's important to consider where your server spends time when a page request comes in. Consider a typical page. It consists of:

- Static content: This is HTML that is determined by the page URL and does not vary from user to user.

- Linked static content: These are static images or other resources that are present on the page, but read through separate requests.
- Dynamic content: This is HTML or other data that is customized for each user.
- Linked dynamic content: These are dynamic images or other resources that are customized for each user and read through separate requests.

Where does a server spend its time when a request comes in?

- Rebuilding the ASP.NET objects
- Loading and sending static content
- Generating and sending dynamic content.

Of these, the biggest performance impact tends to be due to the dynamic content. Static content not only requires no processing beyond loading from disk, it can be efficiently cached as well by both ASP.NET and IIS. Consider your own browsing experience: the greatest delays come from a server responding to a request that is unique to your situation - generating a custom map, purchasing a product, performing an inventory request, waiting for that annoying ad to show up from a separate (and slower) ad banner server...

This leads us to a self-evident and logical conclusion:

> The more customization and processing required to generate a dynamic user page, the slower the response will seem to the user.

And worse, since dynamic content is often generated on outside servers (such as ad servers or increasingly, web services), the response time may not even be under your direct control!

These conclusions are obvious and the situation is unavoidable.

Well, maybe not.

Predictive ASP.NET™

There are some situations where delays due to customization are inevitable. You can't, for example, charge a user for an order before they place it.

But it turns out that there are many applications where these delays are not unavoidable. This is because of the reality of how users actually use web sites.

Users do NOT access web pages randomly.
In a real web site, users tend to follow certain common paths which can be determined through web logs.

Some of these paths are obvious. When a user requests a page, it is virtually certain that they will immediately request all of the images that appear on that page. Others are less obvious and vary from site to site. Yet they are measurable. You can make statements about a site along the lines of "If a user reaches page X, there is a n% chance they will next go to page Y".

It turns out that there is one other important factor that relates to dynamic content. While some customization (such as charging for a purchase) can only take place in response to a user request, many types of customization are in practice optional. For example: ideally, you might wish to customize a banner ad based on past purchase history or known interests of the user. However, you can get away with a generic ad. Other types of data mining based on what you know about a user can generate customized content that is a "bonus" - an opportunity to provide a richer user experience, but not essential to the use of the web application.

Combine these two facts, and suddenly the rules for ASP.NET change.

If you have a high degree of confidence as to what a user is likely to do next on your site.

or

If you would like to create customized content that can improve the user experience, but do not want to make the user wait for it.

then your application can benefit from using Desaware's StateCoder to implement Predictive ASP.NET techniques. What is Predictive ASP.NET? That's what we call our approach for using StateCoder to not only maintain the state of objects between requests, but to perform ongoing processing between requests to prepare dynamic information for the next anticipated user request.

No, we're not Crazy

The idea of Predictive ASP.NET is so contrary to every message you've heard about the advantage of statelessness that you might think we've completely lost our senses. We haven't.

We have nothing against stateless components. They have all of the advantages that Microsoft claims for them. However, we do not believe that statelessness is an all or nothing proposition. Consider images, for example:

When a page includes links to images, you know your server is going to get a request for those images. If they are static images, this is not a problem - because caching will help maintain high performance. But what if they are custom images? Say you are running a photo site and know through a login or cookie who the user is, and wish to extract their photos from a database or retrieve them from an offline web service. Your server is going to have to extract them sooner or later anyway. With StateCoder, rather than waiting for each individual image request to come in, you can start a state machine that continues to work in the background after the initial page request is sent (or even while it is being sent). By the time the individual image requests arrive, StateCoder may well have completed preparing the images, allowing them to be sent immediately to the user. Even if they are not ready, the wait for the user will be shorter.

Or consider an advertising scenario. You may know through experience that once a user hits your site, they will be around for at least another half dozen page views. If you know the user from a previous visit (and have identified them via a Cookie or other scheme), you might want to prepare advertising that is customized for the user. However, this is not essential. So you launch a StateCoder state machine after the first request. This state machine mines your database for past experience with the user and starts generating a selection of custom advertising for the user. If the user's next request comes in before the state machine is ready, you can serve up one of your generic ads immediately without delay. But there's a good chance the dynamic content will be ready, in which case you can provide a fully customized page instantly using the data that has been generated since the last request. You can continue to serve up the customized data, deleting it only after the user has not made a request in a certain amount of time.

You Choose the Tradeoff

Predictive ASP.NET with StateCoder represents a choice and a tradeoff. What you are saying is this:

In return for tying up some additional resources (objects held in memory, and the risk of using some server time to generate content that may never be used) - you gain dramatically improved performance for those using your web application.

And improved performance is a big deal - fast sites are less frustrating, more popular, and tend to keep people around longer. They also become the "first choice" in cases where there are multiple sites to choose from.

How StateCoder handles Predictive ASP.NET

StateCoder is Desaware's new .NET framework for the creation of state machines. For those new to State Machines, refer to the introduction to State machines earlier in this manual and to articles on www.desaware.com. With regards to ASP.NET, each state machine you design implements a sequence of operations. This sequence can perform any operation you wish. It is perfectly suited for asynchronous operations

including other web requests or web service requests. A StateCoder state machine can be stored in an ASP.NET session object *and continue to run in the background*! This is very different from most objects stored in ASP.NET session object which just sit there waiting for the next request. For those system that cannot use session objects (say, with web farms), the Predictive ASP.NET state machines can run on a separate web server which can be accessed regardless of which web server receives a request.

Most important, the StateCoder framework manages its own thread pool and efficiently shares threads among all of the state machines in a process (even if they belong to different sessions). This means that you have complete control over how much of your systems resources are tied up with background operations. If you were to simply create background threads for each session, the impact on your system's performance could be huge. However, by dedicating a limited number of threads for Predictive ASP.NET operations, you can gain the benefits of this approach without jeopardizing the overall performance of your server.

The StateCoderWeb1 and Web1Statemachine sample programs (included with the StateCoder demo and the full StateCoder product), demonstrate how you can use Predictive ASP.NET to dramatically improve the performance of web sites in dynamic content scenarios.

The example simulates a simple online commerce site. In this site, there are two pieces of optional dynamic content - that is, content that we would like to make dynamic, but can be static (or missing) if it is not immediately available.

Each page has a graphic - a logo or banner ad if you will. The default is to provide a static graphic, however a dynamically generated graphic would be preferred (in this example, a graphic containing the user's name). In addition, it would be nice to offer the customer recommendations based on previous purchases.

When the pages load, they check to see if the state machine has completed creating the dynamic data. If not, the user is immediately sent static data

that is already prepared, or the data is simply omitted (as you prefer). If the data is ready, it is sent immediately to the user.

For details on how this is implemented, review the StateCoderWeb1 and Web1StateMachine assemblies.

Reference

This section contains reference information for each of the StateCoder classes. Though it is in the form of a reference, we strongly encourage you to read it carefully as it describes in details how to build state machines using the StateCoder framework.

State Classes

The State and DynamicState classes are the classes on which you build individual states in a state machine.

The State Class

The State class is the base class for every state in a state machine. There are two ways of defining states for a state machine: Static and Dynamic.

The State class is available in two forms, the normal form and the generics form which takes as its parameter the type of the state machine class associated with the state.

```
State(Of class derived from StateMachineBase)
State<class derived from StateMachineBase>
```

In a static state machine, each state is defined by its own class, each derives from the base State class and has its own name. For example:

VB

```
<InitialState()>Public Class myInitialState
Inherits State
End Class
```

C#

```
[InitialState()]class myInitialState:State
      {
```

```
        }
```

The attributes InitialState or FinalState are used to specify which is the initial state and which is the final state for a static state machine. You must define one (and only one) initial state, and one (and only one) final state for each state machine.

The name of the state is the name of the class – in this case the same code defines a state named “myInitialState”.

A state machine that uses static State objects is defined by using the ContainsState attribute on the StateMachine class in this form:

VB

```
<ContainsState("Namespace.someinitialstate"), _
 ContainsState("Namespace.someotherstate"), _
 ContainsState("Namespace.somefinalstate")> _
Public Class SearchMachine
    Inherits StateMachine
End Class
```

C#

```
[ContainsState("Namespace.someinitialstate"),
ContainsState("Namespace.someotherstate"),
ContainsState("Namespace.somefinalstate")]
public class SearchMachine: StateMachine
      {
      }
```

The ContainsState attribute must include the full name of the state (including the namespace). The state name is case sensitive. States can be defined in any order. In static state machines, the individual state objects are created by the framework.

State Object Properties

Machine	VB: Protected Overridable ReadOnly Property Machine() As StateMachineBase
	C#: protected virtual StateMachineBase Machine
	For generic version:
	VB: Protected Overridable ReadOnly Property Machine() As *class derived from StateMachineBase*
	C#: protected virtual *class derived from StateMachineBase* Machine
	
	This property returns a reference to the state machine that owns this state object. When using the generics form of the State class in VS2005, this will return the type of your state machine. When using the normal form or VS2003, you will frequently shadow this method in your state object to provide a reference to your own type of state machine. For example: If your state machine object is of type myStateMachine, you might define the following override:
	`Private Shadows ReadOnly Property Machine() _`
	`As myStateMachine`
	`    Get`
	`        Return CType(MyBase.Machine, _`
	`        myStateMachine)`
	`    End Get`
	`End Property`
	
	If your State object is intended to be used by two different state machines you should avoid using the generic form of the class, and be cautious in creating a shadowed property because it can lead to errors unless the return type is set to an interface that is common to both state machines.

State Object Methods

EnterState	`VB: Public Overridable Sub EnterState()`
	`C#: public virtual void EnterState()`
	
	This method is called by the framework when entering a state. You will usually override this method to perform any necessary tasks. One common task during this method is to set one or more message sources for the state machine.
	
	If your state machine is managed (inherits from the StateMachine class) and there are no active message sources on return from this method, your state machine will abort with an exception. If this method is being called due to a call to the state machine's Start method, the exception will be passed on to the caller of the Start method. If this method is called by the framework, the state machine will go directly to the end state, call the end state's ExceptionReceived method, and set the state machine's LastException property.
	
	When called from a managed state machine, this method is always called on the thread that is running the state machine, except for when it is called for the initial state (in which case it is called on the thread that is creating the state machine).

ExceptionReceived	`VB: Public Overridable Sub ExceptionReceived(ByVal ex As Exception)`
	`C#: public virtual void ExceptionReceived(Exception ex)`
	
	This method is called by the framework when an exception occurs during the operation of a state machine. Refer to the section on State Machines and Exceptions for further details. For managed state machines, this method will be called on the state

	machine's thread regardless of source (the sole exception being exceptions that occur during the EnterState method of the initial state. The default behavior of this method if not overridden is to set the State Machine's LastException property to the exception that occurred and to switch immediately to the Final state. If you override this method to handle an exception directly, be sure to set the State Machine's LastException property if you wish the exception to be registered with the state machine. When called from a managed state machine, this method is always called on the thread that is running the state machine.

MessageReceived	`VB: Public Overridable Sub MessageReceived(ByVal message As Object, ByVal source As IMessageSource)`
	`C#: public virtual void MessageReceived(Object message, IMessageSource source)`
	
	This method is called by the framework when a message is sent to a state. It is up to you to interpret the meaning of messages based on your own state machine and to handle them accordingly – any object can be passed as a message. You will almost always override this method, as it forms the heart of each state machine. The source parameter refers to the message source that sent the message.
	
	This method will never be called for the final state in a state machine.
	
	When called from a managed state machine, this

	method is always called on the thread that is running the state machine

NextState	`VB: Protected Sub NextState(ByVal TheNextState As String)`
	`C#: protected void NextState(String TheNextState)`
	
	This method is called by your state class code in order to switch execution to a different state. The EnterState method of the next state will be called before this method returns. If the EnterState method raises an unhandled exception, this method will catch the exception and call the ExceptionReceived method for that state (not the current state), passing it a reference to the exception object.
	
	Specifying an invalid state will cause an "InvalidState" exception to be raised.

ToString	`VB: Public Overrides Function ToString() As String`
	`C#: public override String ToString()`
	
	Returns the name of the state as defined by the type of the state.
	
	WARNING! Do not override this method. It is left overridable because the DynamicState type must override it. If you override it, the framework will not work correctly.

The DynamicState Class

The DynamicState class derives from the base State class and is the base class for every state in a dynamically defined state machine.

The DynamicState class is available in two forms, the normal form and the generics form which takes as its parameter the type of the state machine class associated with the state.

```
DynamicState(Of class derived from StateMachineBase)
DynamicState<class derived from StateMachineBase>
```

In a dynamic state machine, all of the states are typically implemented by a single class which derives from the base DynamicState class. For example:

[VB]

```
Public Class myDynamicState
Inherits DynamicState
Public Sub New(ByVal StateName As String)
      MyBase(Statename)
End Sub

End Class
```

[C#]

```
public class myDynamicState: DynamicState
{
      public myDynamicState(
            String StateName):base(StateName)
      {

      }
}
```

Each dynamic state is constructed with a reference to a StateMachine or UnmanagedStateMachine object (referring to the actual state machine in which it will be used) and the name of the state. Each state name in a state machine must be unique.

A state machine that uses dynamic state objects is created by passing an array of these state objects to the StateMachine object's constructor. The first object is always considered the initial state. The last object is always considered the final state. The InitialState and FinalState attributes should not be used with dynamic states.

DynamicState Object Properties

Machine	Inherited from State

DynamicState Object Methods

Constructor	`VB: Public Sub New(ByVal StateName As String)`
	`C#: public DynamicState(String StateName)`
	Create dynamic states after you've created your state machine object (derives from StateMachine or UnmanagedStateMachine). Pass the name of the state as the StateName parameter. Each state name in a state machine must be unique.

EnterState	Inherited from State
ExceptionReceived	Inherited from State
MessageReceived	Inherited from State
NextState	Inherited from State. Uses the name specified by the DynamicState constructor to identify states.

ToString	`VB: Public NotOverridable Overrides Function ToString() As String`
	`C#: public override sealed String ToString()`
	Returns the name of the state as defined when the object was constructed.

State Machine Classes

The StateMachineBase, StateMachine and UnmanagedStateMachine classes are the classes on which you build state machines. These classes use state classes to define the states in the state machine.

The StateMachineBase class

The StateMachineBase class forms the foundation on which state machines are built. **Classes you create should not inherit directly from this class!** Instead, your classes should inherit from the StateMachine and UnmanagedStateMachine classes.

Nevertheless, many of the members of the StateMachineBase class are common to both, so they are defined in this section. Only members that are accessible through the StateMachine and UnmanagedStateMachine classes are shown here.

StateMachineBase properties

LastException	`VB: Public Overridable Property LastException() As Exception`
	`C#: public virtual Exception LastException`
	
	The StateCoder framework traps any unhandled exceptions that occur while the state machine is running. Your state machine is notified of these exceptions by the framework, which calls the ExceptionReceived method for the current State object. The most recent exception can also be determined by reading this property.
	
	This property is most often used by external code that is using the state machine to determine what caused a state machine to terminate in the event of an error. This property is set by the ExceptionReceived method of the current state.

	You will typically not override this property, however may do so if you wish to perform additional processing when an exception occurs.

EndStateName	`VB: Public ReadOnly Property EndStateName As String`
	`C#: public String EndStateName`
	
	Returns the name of the final state of this state machine.

Name	`VB: Public Property Name As String`
	`C#: public String Name`
	
	This is a user defined name for the state machine. It is used by the ToString override when returning a string representation for the state machine, and is used during tracing.

StateTransitionMonitor	`VB: Public WriteOnly Property StateTransitionMonitor As StateTransition`
	`C#: public StateTransition StateTransitionMonitor`
	
	Delegate:
	`VB: Public Delegate Sub StateTransition(ByVal EnteringState As String)`
	`C#: public delegate void StateTransition(String EnteringState)`
	
	When set, the specified delegate is called before each state transition. The delegate is called before the EnterState method call on the State object. This event will be raised on the same thread as the EnterState method call. Refer to State Machine Threading for details.
	
	This method is most often called to monitor the

	progress of a state machine. You should NOT use it to modify the behavior of the state machine (switch states, change message sources, etc.). Aside from the fact that doing so violates the principles of encapsulation that state machines are designed to promote, the framework is not designed to handle arbitrary actions during this call.

StateMachineBase methods

AbortStateMachine	`VB: Public Overridable Sub AbortStateMachine()`
	`C#: public virtual void AbortStateMachine`
	
	This method is called to abort a state machine's operation outside of the normal message sequence. The default behavior is to send the StateAbortException to the current state's ExceptionReceived method. If the state machine has the StateMachineFlags.ForceEndStateOnAbort flag set, the state machine will be set to the end state as well.
	
	If you override this method, you should always call the base class method.

GetState	`VB: Protected Overridable Function GetState(ByVal StateName As String) As State`
	`C#: protected virtual State GetState(String StateName)`
	
	Returns the State object based on the state name. States in the framework are always identified by

	name, which consists of either the Type (for State objects) or assigned Name (for DynamicState objects).

Reset	`VB: Public Overridable Sub Reset()`
	`C#: public virtual void Reset()`
	
	This method resets a StateMachine to its initial state. The StateMachine object must currently be in the End state or an EndStateReset StateException exception will be thrown. Any Message Sources set previously in the state machine will have been cleared when the previous state machine finished executing, and will thus need to be set. You must call the Start method to restart the state machine after calling this method

ToString	`VB: Public Overrides Function ToString() As String`
	`C#: public override String ToString()`
	
	Returns the name of the state machine in the form *type (name)* where *type* is the type name of the state machine object, and *name* is the value of the Name property if set. The ToString method is used primarily during tracing.

StateMachineBase events

ReachedEndState	`VB: Public Event ReachedEndState(ByVal sender As Object)`
	`C#: public event ReachedEndStateEventHandler ReachedEndState(Object sender)`
	
	This event is raised by a state machine when it

	enters its end state. The EnterState method of the End State object will be called before this event is raised. The WaitHandle for the state machine will be signaled after this event is raised unless the RaiseEndStateEventAfterSignal state machine flag is specified. For unmanaged state machines, this event will be raised on the same thread as the EnterState method call. Refer to State Machine Threading for details. This event may be overridden (and is overridden by the StateMachine class), but it is unlikely you will need to do so.

IMessageSource Implementation

StateMachineBase objects can serve as message sources to other state machines. To do so, the StateMachineBase object implements the IMessageSource interface. Override these methods to define the way your state machine implements a message source.

MessageReady	`VB: Public ReadOnly Property MessageReady As Boolean`
	`C#: public Boolean MessageReady`
	
	This property is called by other state machines to determine if the state machine has entered the end state. It also provides a general mechanism to determine if a state machine is in the end state.

MessageReadySignal	`VB: Public ReadOnly Property MessageReadySignal() As WaitHandle`
	`C#: public WaitHandle MessageReadySignal`
	
	This property is called by other state machines to retrieve a wait handle which is signaled once a message is ready. The default behavior is to

	return a System.Threading.ManualResetEvent object whose signal state is True if the state machine is in the end state and false otherwise.

MessageSourceOptions	`VB: Public Overridable ReadOnly Property MessageSourceOptions As MessageSourceFlags`
	`C#: public virtual MessageSourceFlags MessageSourceOptions`
	
	This property allows you to specify additional options for the behavior of the state machine when acting as a message source. The default value MessageSourceFlags.OneShot.
	
	This corresponds to the Flags property of the IMessageSource interface.

RetrieveMessage	`VB: Public Overridable Function RetrieveMessage() As Object`
	`C#: public virtual Object RetrieveMessage`
	
	This method may be called by the other state machine to retrieve a message defined by the current state machine. Override this method to specify the message to return. The default is to return Nothing (null).

WaitExpiration	`VB: Public Overridable Property WaitExpiration() As DateTime`
	`C#: public virtual DateTime WaitExpiration`
	
	This property allows you to set a timeout for a message source. The default implementation is to store any value you set. The default member value is Nothing (null). Override this to calculate expiration values on the fly.

The StateMachine Class

The StateMachine class will be the base class for most of your state machines. When you inherit from the StateMachine class, you create a managed state machine – one that is managed by the StateCoder framework. The framework takes care of message dispatch, threading and most synchronization tasks.

StateMachine properties

ActiveMessageSource	`VB: Public WriteOnly Property ActiveMessageSource As IMessageSource`
	`C#: public IMessageSource ActiveMessageSource`
	
	This is a shortcut to setting a single message source for a state machine. See ActiveMessageSources.

ActiveMessageSources	`VB: Public Property ActiveMessageSources As IMessageSource()`
	`C#: public IMessageSource[] ActiveMessageSources`
	
	One or more valid message sources must be defined at all times for the state machine to work (except for when the state machine is in the end state). This property sets and retrieves an array of message sources. Use the ActiveMessageSource property to set a single messages source.
	
	Setting this property clears any existing message sources. Any message sources with MessageSourceFlags set to AutoDispose will be disposed at that time.

LastException	Inherits from StateMachineBase

Name	Inherits from StateMachineBase
StateTranstionMonitor	Inherits from StateMachineBase

StateMachine methods

Constructor	VB: Public Sub New() Public Sub New(ByVal flags As StateMachineFlags) Public Sub New(ByVal flags As StateMachineFlags, ByVal States() As DynamicState) C#: public StateMachine() public StateMachine(StateMachineFlags flags) public StateMachine(StateMachineFlags flags, DynamicState[] States) The flags option allows you additional control over the operation of the state machine (refer to the StateMachineFlags enumeration). The States() array is used to create state machines using dynamic state objects. Refer to the introduction to the State class and Dynamic State class for further details.
AbortStateMachine	VB: Public Overrides Sub AbortStateMachine() C#: public override void AbortStateMachine() This method is called to abort a state machine's operation outside of the normal message sequence. The default behavior is to send the StateAbortException to the current state's

	ExceptionReceived method. If the state machine has the StateMachineFlags.ForceEndStateOnAbort flag set, the state machine will be set to the end state as well. If you override this method, you should always call the base class method. Note that due to synchronization issues, a state machine will not abort immediately.
GetState	Inherits from StateMachineBase
Reset	Inherits from StateMachineBase If you override this method, be sure to call the base class method.

SendException	`VB: Protected Overrides Sub SendException(ByVal ex As Exception)`
	`C#: protected override void SendException(Exception ex)`
	
	This method is called by the framework to send messages to the state classes in the state machine. **You should never call it in your state machine class**.
	
	You may override it (to monitor exceptions), however should be careful to call the base class method if you do so.
	
	This method is always called on the thread that is running the state machine.
	
	If you raise an exception during the processing of this method, it will be ignored.
	
	It is possible for multiple exceptions to be sent to a

	state.

SendMessage	`VB: Protected Overrides Sub SendMessage(ByVal msg As Object, ByVal source As IMessageSource)`
	`C#: protected override void SendMessage(Object msg, IMessageSource source)`
	
	This method is called by the framework to send messages to the state classes in the state machine. **You should never call it in your state machine class.**
	
	You may override it (to monitor messages), however should be careful to call the base class method if you do so.
	
	This method is always called on the thread that is running the state machine.
	
	If you raise an error during the processing of this method, it will be reflected back to a SendException call for the current state (which may be different from the one that executed this method if the NextState method was called).

Start	`VB: Public Overridable Sub Start()`
	`C#: public virtual void Start()`
	
	Call this method to start operation of the state machine. You will typically do this after you have created the state machine and set any message sources (for cases where the messages sources are not set by the State objects).
	
	If you override this method, be sure to call the base class method.
	
	If no message source is active by the time this command returns, an exception will be raised.

ToString	Inherits from StateMachineBase

StateMachine events

ReachedEndState	`VB: Public Event ReachedEndState(ByVal sender As Object)`
	`C#: public event ReachedEndStateEventHandler ReachedEndState(Object sender)`
	
	This event is raised by a state machine when it enters its end state. The EnterState method of the End State object will be called before this event is raised. The StateCoder framework examines all handlers registered for this event. If the handler is derived from the System.Windows.Forms.Control class (which includes all forms and controls), the framework will marshal the event to the control's thread, providing automatic synchronization for these applications.
	
	Event handlers registered to other classes will not be marshaled (use care if you raise an event to a class that has direct access to form or control methods or properties). Handlers to static methods will also not be marshaled. Refer to State Machine Threading for details.

The UnmanagedStateMachine Class

The UnmanagedStateMachine class allows you to use the state machine framework to define state machines that are run entirely under control of your own application. Because the framework is not involved in the actual running of the state machine, it provides no protection relating to thread

safety and no synchronization. All message dispatching must be performed manually. There is no automatic support for message sources.

UnmanagedStateMachine properties

LastException	Inherits from StateMachineBase
Name	Inherits from StateMachineBase
StateTranstionMonitor	Inherits from StateMachineBase

UnmanagedStateMachine methods

Constructor	VB: Public Sub New() Public Sub New(ByVal flags As StateMachineFlags) Public Sub New(ByVal flags As StateMachineFlags, ByVal States() As DynamicState) C#: public UnmanagedStateMachine() public UnmanagedStateMachine(StateMachineFlags flags) public Unmanaged StateMachine(StateMachineFlags flags, DynamicState[] States) The flags option allows you additional control over the operation of the state machine (refer to the StateMachineFlags enumeration). The States() array is used to create state machines using dynamic state objects. Refer to the introduction to the State class and Dynamic State class for further details.

AbortStateMachine	Inherits from StateMachineBase
GetState	Inherits from StateMachineBase
Reset	Inherits from StateMachineBase

SendException	`VB: Protected Overrides Sub SendException(ByVal ex As Exception)`
	`C#: protected override void SendException(Exception ex)`
	This will immediately send an exception to the current state's ExceptionReceived method.

SendMessage	`VB: Protected Overrides Sub SendMessage(ByVal msg As Object, ByVal source As IMessageSource)`
	`C#: protected override void SendMessage(Object msg, IMessageSoruce source)`
	This will immediately send a message to the current state's MessageReceived method.

ToString	Inherits from StateMachineBase

UnmanagedStateMachine events

ReachedEndState	Inherits from StateMachineBase

The StateMachineFlags Enumeration

The StateMachineFlags enumeration is an optional parameter that is passed to the StateMachine and UnmanagedStateMachine constructor to control the behavior of the state machine.

StateMachineFlags enumeration values

None	No flag specified
CreateInNewThread	Only applies to StateMachine objects. This flag indicates that a state machine should run in a private thread. Normally managed state machines run in a thread pool. Private threads should only be used for state machines that perform very long operations synchronously (where processing a message might block the operation of other state machines in the thread).
ForceEndStateOnAbort	When this flag is set, the receipt of a StateAbortException exception by the StateMachineBase class will force the state machine into the end state (which will also cause the end state's EnterState method to be called). If not set, receipt of this exception will simply send the exception to the current state, and cease operation.
ReachedEndStateEvent-AfterSignal	Normally, the ReachedEndStateEvent is raised before the state machine's WaitHandle is signaled. This flag changes the behavior so that the WaitHandle is signaled first. This is necessary if you have a form or control thread both waiting for the state machine

	wait handle and responding to events (which is not recommended anyway because of the risk of race conditions).

Exception Classes

In managed state machines, the StateCoder framework traps exceptions that take place while the state machine is running and send them to the ExceptionReceived method for the current state. This guarantees that exceptions are not only directed to the current state, but are raised in a consistent manner. Refer to the section "State Machines and Exceptions" for more information on how the StateCoder framework handles exceptions.

Exceptions that are generated by the StateCoder framework derive from the StateException class. This class behaves identically to the .NET framework ApplicationException class.

The StateException class

The following exceptions are raised by the framework:

Reset method is only valid for state machines in their end state

Indicates the Reset method was called for an active state machine.

Illegal Parameter Value

Generic message indicating that you passed an invalid parameter to a StateCoder object method or property.

Wrapped object is not valid

Message sources often wrap an internal object – for example: a stream message source contains a stream. This error indicates that the internal object is not valid.

Asynchronous operation is already in progress

Several StateCoder message source objects perform asynchronous operations. This error is raised when you attempt to start an asynchronous operation on an object when one is already in progress.

Duplicate State Name Found - State names must be unique within a state machine.

This exception is raised when you attempt to create a new state machine object that has duplicate states.

Duplicate or missing Initial State

This exception is raised when you define a state machine that has no initial state, or more than one state marked with the InitialState attribute.

Duplicate or missing End state

This exception is raised when you define a state machine that has no end state, or more than one state marked with the FinalState attribute.

Initial and End states must be different

This exception is raised when you define a state machine in which the initial and final state are the same.

Invalid state specified

This exception is raised when an attempt is made to switch to a non-existent state. This will typically occur when you pass an invalid state parameter to the NextState method in a state class.

This exception will also occur during construction of a static state machine if a ContainsState attribute parameter is invalid. You can trap the exception and read the exception's message to find the name of the invalid state.

One common cause of this exception is typographical errors in the state name, including capitalization. Remember – state names are case sensitive.

State Machine has already started

This exception is raised if you attempt to start a state machine that is already running.

Message source missing on start command

This exception is raised if a message source is not specified by the time the state machine's Start method returns. The message source can be specified before the Start method is called, or during the EnterState method of the current state.

Call to StartRead is invalid unless RequiresExplicitStart flag is True

This exception is raised if you call the StartRead method of the ParsingMessageSource class if the RequiresExplicitStart flag was not specified when the source was created.

The specified process does not exist

The process identifier specified in the constructor of the ProcessMessageSource class does not refer to a valid process.

The StateAbortException class

The StateAbortException class is raised when a state machine is aborted. This is usually a result of a call to the AbortStateMachine method of the state machine class. In managed state machines, it can also be called if an application domain terminates while a state machine is active.

The StateTimeoutException class

The StateTimeoutException is raised by the StateCoder framework when a message source times out.

The MessageSource property of this exception object can be used to retrieve a reference to the message source that timed out.

Message Sources

Message sources comprise the third main tier of the StateCoder framework. The StateMachineBase (and descendent) classes represent the state machine as a whole. The State (and descendent) classes represent individual states in the state machine. And Message sources (any class that implements the IMessageSource interface), represent the input to the states that cause them to perform operations and transition from one state to the next.

Managed state machines (those that derive from the StateMachine class) are tightly integrated with message sources. You must make sure that at least one message source is always active between the time the state machine is started (when it returns from the Start method[10]) and the time it enters the end state. Message sources can be set from your state machine class, or from individual state classes (allowing message sources to vary from one state to the next). The framework synchronizes message sources so that messages are always dispatched on the thread on which the state machine is running – avoiding many problems inherent in multithreading applications.

Each managed state machine has a single Active Source list, that comprises all of the message sources that are currently active. Message sources should only be present in one state machine list – they cannot be

[10] The message source may be set before calling the Start method, or may be set within the EnterState method of the initial state.

shared (this is obvious – since it makes no sense for a single message source to be providing messages to two state machines simultaneously).

Unmanaged state machines, in which you dispatch all messages directly to the state machine, provide no automatic support for message sources, however you may create and use them (watching for and then dispatching messages) if you wish.

In addition to defining the standard IMessageSource interface, the StateCoder framework includes a number of useful message sources.

The IMessageSource Interface

Flags	`VB: ReadOnly Property Flags() As MessageSourceFlags`
	`C#: MessageSourceFlags Flags { get; }`
	
	This allows you to specify additional options for how the StateCoder framework will handle the message source when used with managed state machines. Refer to the MessageSourceFlags enumeration for details.
	
	This property has no effect on unmanaged state machines.
	
	You should not raise an exception when this property is read.

MessageReady	`VB: ReadOnly Property MessageReady() As Boolean`
	`C#: Boolean MessageReady { get; }`
	
	This property is used to determine if the message source has a message ready. Return True if a message is ready, False otherwise.

	This property can be read multiple times. However, once the message is actually read using the RetrieveMessage method, this property should either reset to False (if no new message is ready), or remain True (if the next message is ready). You should not raise exceptions when this property is read.

MessageReadySignal	`VB: ReadOnly Property MessageReadySignal() As WaitHandle`
	`C#: WaitHandle MessageReadySignal { get; }`
	
	This property is used to retrieve a wait handle which is signaled once a message is ready. A wait handle is any object that derives from System.Threading.WaitHandle.
	
	Note that multiple message sources are permitted to share wait handles (a useful way to conserve resources, allowing a framework that manages multiple message sources to use one wait handle to signal when any of the sources is ready). This means that you should always use the MessageReady property to check if a message is ready, and not rely only on the wait handle.
	
	If you are using a message source with unmanaged state machines, keep in mind that the .NET wait functions do not permit duplicate wait handles. The StateCoder framework deals with this situation automatically with managed state machines.
	
	You MUST return a valid WaitHandle object

	when this method is called. Any exception raised when processing this method is considered fatal and will cause the state machine to abort.

RetrieveMessage	`VB: Function RetrieveMessage() As Object`
	`C#: Object RetrieveMessage()`
	
	This method is called to retrieve a message. A message can be any arbitrary object as defined by the message source. It is up to you to make sure that your state machine properly interprets and handles incoming messages.
	
	Once a message is retrieved, the message source should either be ready to return the next message, or should start (or be prepared to start) another message retrieval operation. Each message may be retrieved only once with this property.
	
	Note that some message sources, such as timer alarms, may not actually return a message – in those cases the fact that the source is ready provides sufficient information to the state machine.
	
	With managed state machines, any errors raised during this method will be reflected back to the ExceptionReceived method of the current state.

WaitExpiration	`VB: Property WaitExpiration() As DateTime`
	`C#: DateTime WaitExpiration {get; set; }`
	
	This allows you to set a timeout for a message source. You may return DateTime.MinValue (zero) to indicate that no expiration time is set.
	
	With managed state machines, you should set timeout values at the same time as you set the message source. Changing the WaitExpiration

	value while the framework is waiting for messages will not change the expiration (i.e., once the framework starts waiting on a message source, the expiration time is already set). The expiration of a message source will cause a StateTimeoutException to be sent to the current state's ExceptionReceived method. This property has no effect with unmanaged state machines, though you may of course use it if you wish. You should not raise an exception when this property is read.

The MessageSourceFlags Enumeration

The MessageSourceFlags enumeration provides additional information to the StateCoder framework as to how message sources should be handled. The value is returned by the Flags property of the IMessageSource interface.

MessageSourceFlags Values

None	0 – No flags specified
OneShot	When the StateCoder framework sees a message source with this option set, it automatically removes it from the Active Source list for the state machine after a message is processed. This option is designed for message sources, such as state machines, that are intended to produce only a single message and then be destroyed.
AutoDispose	This indicates that a message source implements

	the IDisposable interface and needs to be disposed. It also indicates that the message source should be used only once and that you wish the framework to Dispose it automatically when it is removed from the Active Source list for a state machine.

The AlwaysSignaledWaitHandle Class

The StateCoder framework uses wait handles to suspend a thread if none of the state machines in a thread have a message ready. Thus, generally speaking, if you define a message source that was always able to provide a message (which is not at all uncommon – a text processing state machine would fall into this category), you might think it would never actually need to provide a wait handle. The framework would always see that it's MessageReady property is True, and would thus never request a wait handle from the MessageReadySignal property.

However, it turns out that it is necessary for message sources that are always ready to nonetheless return a wait handle. This is for two reasons:

1. The IMessageSource.MessageReadySignal property is specified as always returning a valid wait handle.
2. In practice, it is possible for a state machine to be added to a thread right after the framework decides it needs to suspend a thread. In which case a wait handle may be requested from the message source.

The AwaysSignaledWaitHandle class implements the shared (static) method Handle, that returns a ManualResetEvent object that is always signaled.

```
Public Shared ReadOnly Property Handle() As WaitHandle
```

You should use the handle retrieved from this method in any case where your message source needs to return a signaled object. This significantly reduces the resource load on the system. Remember, the StateCoder framework allows the same wait handle to be shared among multiple message sources.

The GenericMessageSourceBase Class

The GenericMessageSourceBase class provides an efficient base class implementation for all message sources that do not generate their own wait handles, or that wrap internal objects that do not generate their own wait handles.

GenericMessageSourceBase members

Dispose	`VB: Public Overridable Sub Dispose() Implements IDisposable.Dispose`
	`C#: public override void IDisposable.Dispose`
	
	The GenericMessageSourceBase class implements IDisposable in order to close the ManualResetEvent wait handle. You should call Dispose on this class when you are through with it. You may override the Dispose method if your message source has cleanup that needs to be done, in which case you should always call the base class Dispose method as well.

Finalize	`VB: Protected Overrides Sub Finalize()`
	`C#: ~GenericMessageSourceBase`
	
	The GenericMessageSourceBase class implements Finalize in order to close the ManualResetEvent wait handle. You should call Dispose on this class when you are through with it. You may override the Finalize method if your

	message source has cleanup that needs to be done, in which case you should always call the base class Dispose method during the Finalize event. Objects that are Disposed will not also be finalized.

SourceFlags	`VB: Public Overridable Property SourceFlags As MessageSourceFlags`
	`C#: public virtual MessageSourceFlags SourceFlags`
	
	You may override the default message source flags. Or you may set the source flags for an individual instance of a message source. The default value is zero.

MessageReady	`VB: Public MustOverride ReadOnly Property MessageReady As Boolean public Boolean`
	`C#: public abstract Boolean MessageReady`
	
	You must override this property to indicate whether a message is ready for your message source. Be sure to call the SetExistingWaitHandle method any time the MessageReady state for your message source changes.

MessageReadySignal	`VB: Public ReadOnly Property MessageReadySignal As WaitHandle`
	`C#: public WaitHandle MessageReadySignal`
	
	The GenericMessageSourceBase class returns a manual reset event whose signaled value depends on the value of the MessageReady property (which you have overridden).

RetrieveMessage	`VB: Public MustOverride Function RetrieveMessage() As Object`
	`C#: public abstract Object RetrieveMessage`
	
	You must override this method to return a message from your message source.

SetExistingWaitHandle	`VB: Protected Sub SetExistingWaitHandle(ByVal signaled As Boolean)`
	`C#: protected void SetExistingWaitHandle(Boolean signaled)`
	
	Use this method to control the signal state of the wait handle for your message source. Typically you will call it with the signaled parameter set to True any time a message is ready, and with the signaled parameter set to False any time a message is no longer ready (often after the message is read using the RetrieveMessage method).

WaitExpiration	`VB: Public Overridable Property WaitExpiration() As DateTime`
	`C#: public virtual DateTime WaitExpiration`
	
	You may override this method to provide your own expiration timeouts for your message source. The default implementation simply treats this as a variable you can set. The default value is no timeout.

For further information on using this class, refer to the tutorial on building custom message sources.

The AsyncResultMessageSource Class

The AsyncResultMessageSource class makes it easy to use virtually any asynchronous operation as a message source. This class allows you to perform most asynchronous operations without creating a custom message source for each one. The following code shows a typical scenario, in this case for a web request.

[VB]

```
httpwr = CType(WebRequest.Create(newuri), _
HttpWebRequest)
' Start the async request
msource = New AsyncResultMessageSource( _
httpwr.BeginGetResponse( _

AsyncResultMessageSource.GetAsyncCallbackFunction() _
, Nothing))
' Set the request as the message source
ActiveMessageSource = msource
```

[C#]

```
httpwr = (HttpWebRequest)(WebRequest.Create(newuri));
                          // Start the async request
msource = new
AsyncResultMessageSource(httpwr.BeginGetResponse(
AsyncResultMessageSource.GetAsyncCallbackFunction(),
null));
// Set the request as the message source
ActiveMessageSource = msource;
```

The AsyncResultMessageSource class provides a generic AsyncCallback delegate that can be used with most .NET asynchronous operations, thus eliminating the need to define your own delegate to handle asynchronous operations. When the asynchronous operation is complete, the AsyncResultMessageSource object will be ready and will return the internal IAsyncResult object as the message which can then be processed as necessary to end the asynchronous operation as shown here:

[VB]

```
Public Overrides Sub MessageReceived(ByVal message As
Object, _
ByVal source As IMessageSource)
iar = CType(message, IAsyncResult)
webresponse = CType(httpwr.EndGetResponse(iar),
HttpWebResponse)
```

[C#]

```
public override void MessageReceived(object message,
IMessageSource source){
IAsyncResult iar;
iar = (IAsyncResult)message;
webresponse = (HttpWebResponse)
(Machine.httpwr.EndGetResponse(iar));
```

The AsyncResultMessageSource class correctly handles asynchronous operations that complete synchronously immediately when started, thus allowing it to be used as a generic solution.

AsyncResultMessageSource members

AsyncCallbackFunction	`VB: Public Overridable Sub AsyncCallbackFunction(ByVal ar As IAsyncResult)`
	`C#: public virtual void AsyncCallbackFunction(IAsyncResult ar)`
	This method can be overridden if you are creating a derived class and wish to perform some operation during the actual asynchronous delegate call rather than waiting for the message to be processed.
	This method is a “stub” and has no default behavior.

Constructor	`VB: Public Sub New(ByVal Async As IAsyncResult)` `public C#:`

	`AsyncResultMessageSource(IAsyncResult Async)` The constructor for this class requires an IAsyncResult value that is the result of launching an asynchronous operation.

Dispose	`VB: Public Overridable Sub Dispose() Implements IDisposable.Dispose`
	`C#: public virtual void IDisposable.Dispose()`
	
	The AsyncResultMessageSource class implements IDisposable in order to clear internal resources. You should call Dispose on this class when you are through with it. You may override the Dispose method if your message source has cleanup that needs to be done, in which case you should always call the base class Dispose method as well.

Finalize	`VB: Protected Overrides Sub Finalize()`
	`C#: ~AsyncResultMessageSource`
	
	The AsyncResultMessageSource class implements Finalize in order to clear internal resources. You should call Dispose on this class when you are through with it. You may override the Finalize method if your message source has cleanup that needs to be done, in which case you should always call the base class Dispose method during the Finalize event.
	
	Objects that are Disposed will not also be finalized.

Flags	VB: Public Overridable ReadOnly Property Flags As MessageSourceFlags Implements IMessageSource.Flags C#: public MessageSourceFlags IMessageSource.Flags This property returns the MessageSourceFlags value for this message source. The base class returns both MessageSourceFlags.OneShot and MessageSourceFlags.AutoDispose in recognition of the fact that an asynchronous operation is an operation that produces a single result.
GetAsyncCallbackFunction	VB: Public Shared Function GetAsyncCallbackFunction() As AsyncCallback C#: public static AsyncCallback GetAsyncCallbackFunction() Use this method to retrieve an AsyncCallback delegate which you can pass to methods that start asynchronous operations. This eliminates the need to define separate methods and delegates for different asynchronous operations.
MessageReady	VB: Public ReadOnly Property MessageReady As Boolean Implements IMessageSource.MessageReady C#: public Boolean IMessageSource.MessageReady This property returns True once the asynchronous operation is complete or if it completes synchronously.

MessageReadySignal	`Public ReadOnly Property MessageReadySignal() As WaitHandle Implements IMessageSource.MessageReadySignal` This property returns the wait handle for the asynchronous operation. A signaled wait handle is returned if the operation completed synchronously.

RetrieveMessage	`VB: Public Overridable Function RetrieveMessage() As Object Implements IMessageSource.RetrieveMessage`
	`C#: public virtual Object IMessageSource.RetrieveMessage()`
	
	The default implementation of this method returns the original IAsyncResult object passed to the constructor of this object.
	
	Override this to return a different message if you are creating a derived class.

The ManualMessageSource class

The ManualMessageSource class is a sealed (not inheritable) message source that allows you to send messages from any external code into a state machine. The class queues messages, and handles all necessary synchronization.

The ManualMessageSource members

Dispose	Inherits from GenericMessageSourceBase
SourceFlags	Inherits from GenericMessageSourceBase
MessageReady	`VB: Public Overrides ReadOnly Property MessageReady As Boolean` `C#: public override Boolean MessageReady`

	Returns True if a message is available in the messages source's internal message queue.
MessageReadySignal	Inherits from GenericMessageSourceBase

RetrieveMessage	`VB: Public MustOverride Function RetrieveMessage() As Object`
	`C#: public abstract Object RetrieveMessage()`
	
	Retrieves a message from the queue if one is present.

SendMessage	`VB: Public Sub SendMessage(ByVal obj As Object)`
	`C#: public void SendMessage(Object obj)`
	
	Call this method to add a message to the message source queue.

The AlarmMessageSource class

The AlarmMessageSource class is a sealed (not inheritable) message source that wraps a timer. It handles both single alarm and periodic timer events. This class inherits from the GenericMessageSourceBase class.

The AlarmMessageSource members

Constructor	`VB: Public Sub New(ByVal AlarmTimer As TimeSpan, ByVal IntervalTimer As TimeSpan)`
	`Public Sub New(ByVal AlarmTimer As TimeSpan, ByVal IntervalTimer As TimeSpan, Flags as MessageSourceFlags)`
	`C#: public void AlarmMessageSource (TimeSpan AlarmTimer, TimeSpan IntervalTimer)`
	`public void AlarmMessageSource (TimeSpan AlarmTimer, TimeSpan IntervalTimer, MessageSourceFlags Flags)`

	The constructor takes an alarm value and Interval value. Refer to the documentation for the System.Threading.Timer class for details on how these parameters work (they are passed from this constructor directly to the wrapped Timer object)[11]. The MessageSourceFlags allows you to specify flags for this message source.

Change	`VB: Public Sub Change(ByVal AlarmTimer As TimeSpan, ByVal IntervalTimer As TimeSpan)`
	`C#: public void Change(TimeSpan AlarmTimer, TimeSpan IntervalTimer)`
	
	This method changes the timer values. Refer to the documentation for the System.Threading.Timer.Change method for details on the how these parameters work (they are passed from this method directly to the wrapped Timer object).

Dispose	Inherits from GenericMessageSourceBase
SourceFlags	Inherits from GenericMessageSourceBase

MessageReady	`VB: Public Overrides ReadOnly Property MessageReady() As Boolean`
	`C#: public override Boolean MessageReady`
	
	Returns True if the internal timer has elapsed.

MessageReadySignal	Inherits from GenericMessageSourceBase
RetrieveMessage	`VB: Public Overrides Function RetrieveMessage() As Object` `C#: public override Object`

[11] Reminder: The first constructor of the TimeSpan class takes an integer value that specifies ticks in units of 100ns. Multiply by 10,000 to specify milliseconds.

	`RetrievMessage()` Returns True if the timer has elapsed. False otherwise. Calling this method also resets the timer. If the timer is set up to be a periodic timer, the MessageReady property is set to False and the message source is set up to wait for the next timer event.
WaitExpiration	Inherits from GenericMessageSourceBase

The ProcessMessageSource class

The ProcessMessageSource class is a sealed (not inheritable) message source that signals and remains signaled when a specified process exits. This class inherits from the GenericMessageSourceBase class.

The ProcessMessageSource members

Constructor	`VB: Public Sub New(ByVal ProcessID As Integer)`
	`Public Sub New(ByVal ProcessFileName As String)`
	`Public Sub New(ByVal ProcessFileName As String, ByVal ProcessCmdLine As String)`
	`C#: public void ProcessMessageSource (int ProcessID)`
	`public void ProcessMessageSource (string ProcessName)`
	`public void ProcessMessageSource (string ProcessName, string ProcessCmdLine)`
	
	The ProcessID parameter is the process number of a running process you want to wait on. ProcessFileName and ProcessCmdLine is used for cases where you want to launch a new process and wait for that process to exit. ProcessFileName is the file name of the process

	to run, include the full path of the file if necessary. ProcessCmdLine is a string containing any additional command line arguments to pass to the process to launch.
Dispose	Inherits from GenericMessageSourceBase
SourceFlags	Set to AutoDispose and OneShot.

MessageReady	`VB: Public Overrides ReadOnly Property MessageReady() As Boolean`
	`C#: public override Boolean MessageReady`
	
	Returns True if the process has exited. After the RetrieveMessage function is called, will return False.

MessageReadySignal	Inherits from GenericMessageSourceBase

RetrieveMessage	`VB: Public Overrides Function RetrieveMessage() As Object`
	`C#: public override Object RetrievMessage()`
	
	Returns the internal System.Diagnostics.Process object. You can retrieve additional information for the process through the exposed functions. Note certain Process properties are not accessible after the process has terminated.

WaitExpiration	Inherits from GenericMessageSourceBase

The ParsingStreamReader class

The ParsingStreamReader class turns any stream into a message source. In doing so, it performs a number of useful tasks:

- Allows a state machine to begin processing input from a stream before all of the data has been retrieved (in the case of asynchronous operations).
- Allows a state machine to process input on streams such as Sockets where the content determines when data can be processed.
- Allow you to break up stream data into messages based on either a predefined set of rules, a regular expression, or a custom parser.

The ParsingStreamReader handles any text based stream.

With it's ability to parse incoming stream data, the ParsingStreamReader is incredibly useful even outside of state machines!

The ParsingStreamReader class is sealed (not inheritable).

The ParsingStreamReader Members

Constructor	`VB: Public Sub New(ByVal BaseStream As Stream, ByVal Parser As IMessageParser, ByVal flags as ParsingStreamReaderFlags)`
	`Public Sub New(ByVal BaseStream As Stream, ByVal UseEncoder As Text.Encoding, ByVal Parser As IMessageParser, ByVal flags as ParsingStreamReaderFlags)`
	`C#: public ParsingStreamReader(Stream BaseStream, IMessageParser Parser, ParsingStreamReaderFlags flags)`
	`public ParsingStreamReader(Stream BaseStream, Text.Encoding UseEncoder, IMessageParser Parser, ParsingStreamReaderFlags flags)`
	
	The BaseStream parameter defines the stream that is wrapped by the ParsingStreamReader. The UseEncoder parameter specifies the encoder to use for the string (the default is

	UTF8Encoding). The Parser parameter specifies an object that implements the IMessageParser interface (which will be defined later). The flags parameter allows you to specify one or more of the following flags from the ParsingStreamReader enumeration: RequiresExplicitStart – The stream reader must be started with an explicit call to the StartRead method and stopped with an explicit call to the StopRead method. QueueFinalChars – Any leftover (unmatched) text in the stream when the end of the stream is reached, will be treated as a message. When not set, leftover text is discarded.
Dispose	Inherits from GenericMessageSourceBase **Note!** Calling Dispose on the ParsingStreamReader does NOT close or Dispose the underlying stream.
Finalize	Inherits from GenericMessageSourceBase
SourceFlags	Inherits from GenericMessageSourceBase. The default return value is zero.

MaxInternalQueueSize	`VB: Public Property MaxInternalQueueSize As Integer`
	`C#: public int MaxInternalQueueSize`
	
	The ParsingStreamReader parses incoming string data as it arrives and stores individual messages in an internal queue. This property specifies the maximum number of items to store in the queue. Once the queue is full, parsing will stop until enough messages are read to reduce

	the queue size below this threshold. While not parsing data, incoming information will be buffered.

MessageReady	`VB: Public ReadOnly Property MessageReady() As Boolean`
	`C#: public Boolean MessageReady {get;}`
	
	Returns True if a message is available in the messages source's internal message queue.

MessageReadySignal	Inherits from GenericMessageSourceBase

RetrieveMessage	`VB: Public Function RetrieveMessage() As Object`
	`C#: public Object RetrieveMessage()`
	
	Retrieves a message from the queue if one is present. Retrieves a null object (Nothing) when the end of the stream is reached.

StartRead	`VB: Public Sub StartRead()`
	`C#: public void StartRead()`
	
	Use when the RequiresExplicitStart flag is set to begin the stream read operation.

StopRead	`VB: Public Sub StopRead()`
	`C#: public void StopRead()`
	
	Call this method to abort a read operation. When the RequiresExplicitStart flag is set, this method is called to notify the reader that the end of file has been reached.

WaitExpiration	Inherits from GenericMessageSourceBase

The RequiresExplicitStart flag is used in cases where you may be writing data into a stream in segments, each of which, when read, will cause an end of file indication. A good example of this is the CommandLine example, where the same console stream is reused and an end of file is

reached at the end of each line. In that case the StartRead and StopRead methods are used for each read operation on a given stream.

The IMessageParser interface

This interface contains the single method:

```
VB: Sub Parse(ByVal que As Queue, ByRef Chars() As
Char)

C#: void Parse(Queue que, Char[] Chars)
```

Each time it is called, it removes as many characters as possible from the Chars() array, parsing them into individual messages that are placed on the Queue object.

The ParsingClass Class

The StateCoder framework includes the ParsingClass class, which implements the IMessageParser interface.

The ParsingType enumeration defines the type of parsing supported by this class as follows:

```
LineBreak = 1
RegularExpression = 98
Custom = 99
```

The LineBreak option parses the incoming text into lines, where each line consists of a message. With this option, a CRLF pair, or standalone CR or standalone LF are considered line breaks.

The RegularExpression option parses the incoming text according to a Regular Expression expression. You can learn more about Regular Expressions in Dan Appleman's PDF-EBook "Regular Expressions with .NET"

The Custom option allows you to specify a delegate to do custom parsing of the incoming text.

The ParsingClass Members

Constructor	`VB: Public Sub New(ByVal ParseType As ParsingType)`
	`C#: public void ParsingClass(ParsingType ParseType)`
	
	Use this constructor to specify parsing types that do not require additional parameters. At this time only ParsingType "LineBreak" is supported with this constructor.

Constructor	`VB: Public Sub New(ByVal Pattern As String, Optional ByVal options As RegexOptions = RegexOptions.None)`
	`C#: public void ParsingClass(String Pattern, RegexOptions options)`
	
	The Pattern represents the Regular Expression to use for parsing. Each Regular Expression Match is considered a message. Unmatched text is discarded. The RegexOptions parameter specifies the RegexOptions to use for the search.

Constructor	`VB: Public Sub New(ByVal CustomParser As ParseNextFunction)`
	`C#: public void(ParseNextFunction CustomParser)`
	
	Use this constructor to specify a custom parsing function. The custom parsing function must match the delegate:
	
	`VB: Public Delegate Function ParseNextFunction(ByVal Chars() As Char, ByVal StartLoc As Integer, ByRef NextStart As Integer) As Integer`

	`C#: public delegate int ParseNextFunction(Char[] Chars, int StartLoc, ref int NextStart)` The function should start scanning the text at the location specified by StartLoc, and set the NextStart parameter to the location of the start of the next message. The ParsingClass object will extract the message from the Chars() array and add it to the queue.

The ParsingClass implements the IMessageParser interface and can be passed as a parameter to the constructor of the ParsingStreamReader class.

The QueuedStream class

The QueuedStream class is a custom stream class that is designed to be particularly useful with the ParsingStreamReader class. It is very similar to a MemoryStream class, except for the following:

- Data is always written synchronously into the stream.
- Data can be retrieved synchronously or asynchronously from the stream (in other words, the stream will block or wait for data to be written, much like a pipe would).
- Data is always read first in, first out.
- The QueuedStream class is thread safe for multiple writer, single reader operations.

The QueuedStream class is demonstrated in the CommandLine sample application.

The QueuedStream Members

Constructor	`VB: Public Sub New()`
	`C#: public QueuedStream()`
	See the parameterless constructor for the MemoryStream class for more information..

SyncRoot	`VB: Public ReadOnly Property SyncRoot As Object`
	`C#: public Object SyncRoot {get;}`
	An object you can use for SyncLock operations in derived classes.

IsClosed	`VB: Public ReadOnly Property IsClosed As Boolean`
	`C#: public bool MaxInternalQueueSize {get;}`
	True if the stream has been closed.

CanRead	Inherits from Stream. This property is always True.
CanSeek	Inherits from Stream. This property is always False.
CanWrite	Inherits from Stream. This property is always True.

Length	`VB: Public ReadOnly Property Length As Integer`
	`C#: public int Length {get;}`
	Returns the number of bytes currently in the stream.

The remaining members are identical to the MemoryStream class with the following critical exceptions:

All Read operations on a MemoryStream are synchronous, and if data is unavailable the function returns immediately with no data (indicating end of stream).

With the QueuedStream class, an empty stream is considered “waiting” for data (much like a NetworkStream). The QueuedStream class will block (or in the case of an async read, wait) until data becomes available or the stream is closed.

All data is appended to the stream. All data is read from the beginning of the stream.

FrameWork control

The StateCoder framework exposes additional methods that allow you to customize the behavior of the framework. These consist of static methods of the StateManager class.

StateManager static methods

MaxThreadPoolSize	`VB: Public Shared Property MaxThreadPoolSize As Integer`
	`C#: public static int MaxThreadPoolSize`
	
	This property specifies the maximum number of threads that will be created on the StateCoder thread pool for a given process. The default number is 4. The legal range is 1 to 200 (though 200 is far more than is practical for most applications).
	
	You can reduce the size of the thread pool at any time, however threads will only be removed through attrition as the state machines they run terminate naturally. Thus the change will not take place immediately or in a deterministic time

	frame. This property has no impact on the number of threads that may be created for state machines that are instructed to run in their own thread.

MachinesPerThreadThreshold	`VB: Public Shared Property MachinesPerThreadThreshold() As Integer`
	`C#: public static int MachinesPerThreadThreshold`
	
	This property indicates the number of state machines that should exist in a thread before the framework considers creating a new thread. The StateCoder framework does not create new threads unless all of the current threads in the thread pool have at least this many state machines active. Note that under no circumstances will a thread be created if that would cause the thread pool size to increase beyond the value of the MaxThreadPoolSize property. Thus you might have more state machines in a thread than specified by this property. The default value for this property is 8. There is no maximum value for this property.

The StateManager class has additional public properties and methods that are used by the framework. They are not intended to be called from user code. **Doing so may cause the framework to fail to operate correctly and is definitely unsupported.**

Diagnostic Classes

Debugging state machines poses its own set of challenges, especially when you have many states, or many state transitions. The StateCoder framework therefore provides sophisticated tracing capability that is integrated into the .NET framework's diagnostic and instrumentation system. You can read more about this system in the section "State Machine Tracing and Diagnostics."

The SCTraceSwitch class

The SCTraceSwitch class inherits from the base System.Diagnostics.Switch class. The display name for configuration settings is "StateCoderSwitch", the description is "Statecoder state machine tracing".

This class defines the public TraceOptions enumeration that defines the types of tracing you wish to enable and allows you to fine tune the information that is captured. The tracing used is the logical Or of the TraceOptions values.

TraceOptions Enumeration Values

Off = 0	Tracing is disabled
Undefined = 0	Tracing is disabled
EnterState = 1	Tracing occurs immediately before the EnterState method of the current state is called.
MessageReceived = 2	Tracing occurs immediately before the MessageReceived method of the current state is called to process a message.
StateTransition = 4	Tracing occurs on state transitions. The trace will occur before both the EnterState method call, and the StateTransitionMonitor hook if one is installed.

ExceptionReceived = 8	Tracing occurs immediately before the ExceptionReceived method of the current state is called to process an exception.
FinalState = 16	Tracing occurs after the state machine has entered the final state (after the EnterState method for the final state has been called, and after the state machine's end state event has been raised and wait handle has been signaled).
Reset = 32	Tracing occurs when the state machine Reset method is called.
All = 255	All tracing is enabled.

SCTraceSwitch Class Members

Level	`VB: Public Property Level() As TraceOptions`
	`C#: public TraceOptions Level`
	
	This property sets and retrieves the current TraceOption enumeration value that defines the tracing level.

OnEnterState	`VB: Public ReadOnly Property OnEnterState() As Boolean`
	`C#: public Boolean OnEnterState`
	
	This property returns True if EnterState tracing is enabled.

OnMessageReceived	`VB: Public ReadOnly Property OnMessageReceived() As Boolean`
	`C#: public Boolean OnMessageReceived`
	
	This property returns True if MessageReceived tracing is enabled.

OnStateTransition	`VB: Public ReadOnly Property OnStateTransition() As Boolean` `C#: public Boolean OnStateTransition`

	This property returns True if StateTransition tracing is enabled.

OnExceptionReceived	`VB: Public ReadOnly Property OnExceptionReceived() As Boolean`
	`C#: public Boolean OnExceptionReceived`
	
	This property returns True if ExceptionReceived tracing is enabled.

OnFinalState	`VB: Public ReadOnly Property OnFinalState() As Boolean`
	`C#: public Boolean OnFinalState`
	
	This property returns True if FinalState tracing is enabled.

OnReset	`VB: Public ReadOnly Property OnReset() As Boolean`
	`C#: public Boolean OnReset`
	
	This property returns True if Reset tracing is enabled.

The StateCoderTraceEvent class

While most tracing situations rely on simple text messages written to a debugger, the StateCoder tracing system is designed to capture a great deal of information that can then be analyzed using any tool able to read a data table (such as a database or Excel).

The StateCoderTraceEvent class hold the information for a single trace event.

StateCoderTraceEvent Members

CurrentState	`VB: Public ReadOnly Property CurrentState As String` `C#: public String CurrentState`

	This returns the state name of the current state when this event is raised. If a state transition event occurred, this will be the new state.

EventType	`VB: Public ReadOnly Property EventType As String`
	`C#: public String EventType`
	
	This returns the TraceOptions enumeration name for this event. For example “StateTransition” or “MessageReceived”

ExceptionDescription	`VB: Public ReadOnly Property ExceptionDescription As String`
	`C#: public String ExceptionDescription`
	
	This returns the exception for ExceptionReceived events. The string takes the form A:B, where A is the result of the ToString method on the exception object, and B is the exception message.

MachineName	`VB: Public ReadOnly Property MachineName As String`
	`C#: public String MachineName`
	
	This returns the name of the state machine that generated this event. This is derived from the state machine’s Name property.

MachineType	`VB: Public ReadOnly Property MachineType As String`
	`C#: public String MachineType`
	
	This returns the class name of the state machine that generated this event.

Message	`VB: Public ReadOnly Property Message As String`
	`C#: public String Message`
	
	This returns the string representation of the message (using the ToString method of the message) that generated this event (valid for

	MessageReceived events).

MessageSource	`VB: Public ReadOnly Property MessageSource As String`
	`C#: public String MessageSource`
	
	This returns the string representation of the message source (using the ToString method of the message source) that generated this event (valid for MessageReceived events).

ToString	`VB: Public Overrides Function ToString() As String`
	`C#: public override String ToString`
	
	This override provides a detailed English text description of the event.

The output format of trace information in the standard debug window (which is produced by the ToString method of the TraceEvent class) is as follows:

statemachine [(*machinename*)] *message* [*currentstate*] [*details*]

statemachine := The class name of the state machine class.

machinename := The value of the Name property for the state machine if available.

message := A description of the trace event.

currentstate := If applicable, the name of the current state.

details := Additional details about the event.

The SCTraceListener class

The SCTraceListener class inherits from the base System.Diagnostics.TraceListener class. The default listener simply displays a string description of each event (using the ToString method for each SCTraceEvent object). The SCTraceListener builds a database of events which can be accessed in your application or can dump a CSV file format table for later examination using the tools of your choice.

The table format defined by the Listener consists of the following fields:

- EventType
- MachineName
- MachineType
- CurrentState
- MessageSource
- Message
- Exception

These fields all contain strings and correspond to the matching properties of the StateCoderTraceEvent object.

The SCTraceListener class only records trace events marked as belonging to the StateCoder category.

SCTraceListener Members

Constructor	`VB: Public Sub New()`
	`C#: public void SCTraceListener()`
	
	Constructs the listener and initializes the internal DataSet into which events will be stored.

GetData	`VB: Public ReadOnly Property GetData As DataSet` `C#: public DataSet GetData`

	Retrieves a DataSet containing all traced events. The DataSet contains a single DataTable object with the fields described previously.

MachineNameFilter	`VB: Public Property MachineNameFilter As String`
	`C#: public String MachineNameFilter`
	
	If set, only entries that have a machine name where the machine name matches this value, are stored.

MachineTypeFilter	`VB: Public Property MachineTypeFilter As String`
	`C#: public String MachineTypeFilter`
	
	If set, only entries that have a machine type (class name) where the machine type matches this value, are stored.

Write	`VB: Public Overloads Overrides Sub Write(ByVal message As String)`
	`C#: public override void Write(String message)`
	
	Called by the tracing system, this method has no effect.

Write	`VB: Public Overloads Overrides Sub Write(ByVal message As Object, ByVal category As String)`
	`C#: public override void Write(Object message, String category)`
	
	Called by the tracing system, this method records events generated by the framework. Only events belonging to the “StateCoder” category and messages of type StateCoderTraceEvent are recorded.

WriteLine	See Write and WriteLine

Dispose	`VB: Protected Overloads Overrides Sub Dispose(ByVal IsDisposing As Boolean)`
	`C#: protected override void Dispose(Boolean IsDisposing)`
	
	You should dispose the listener when you are finished with it.

DumpCSV	`Public Sub DumpCSV(ByVal outputstream As IO.Stream)`
	`public void DumpCSV(IO.Stream outputstream)`
	
	Dumps the current contents of the DataSet in the CSV format to the specified stream. The DataSet is not cleared by this operation.

State Machine Threading Issues

One of our original purposes in developing StateCoder related directly to the problem of multithreading and asynchronous programming. The fact of the matter is that anybody who claims that multithreaded programming is easy has never done it. It is difficult, and while it can provide many benefits, it can also lead to bugs that are difficult to find and correct. In fact, when you see a Windows application freeze or crash, odds are pretty good that you've just run into a threading bug, one that occurs so rarely that the developers have not been able to reproduce it reliably enough to correct it.

When Desaware started writing VBX control (oh so many years ago), we first wrote a C++ class framework for those controls (though we never commercialized it, it was the first C++ class framework for VBX controls written). So it was that when we looked at the kinds of products we wanted to do for .NET, we decided we needed a strong framework for multithreaded and asynchronous operations on which to build our future products.

But this time we decided to share this technology in a commercial product.

If you've read through this manual, you already have a good understanding of how threading works in StateCoder. But here is a brief summary of the issues that you should keep in mind.

All calls into a state machine are on the state machine's thread, except...

The StateCoder framework assigns each state machine to a single thread (either on the thread pool[12], or a dedicated thread, depending on the flag settings when the state machine is created. All messages to each state will arrive on that same thread. This dramatically reduces the risk of

[12] Each Application Domain has its own StateCoder thread pool.

multithreading problems. However, there are some exceptions and potential problems that you should be aware of:

- The EnterState method of the initial state is called on the thread that calls the Start method of the state machine. If you call the NextState method during the EnterState method of the initial state, that state's EnterState method (and any subsequent states called in this matter) will also run on that thread. However, the state machine will not actually start running until the state machine's Start method call returns, so there is no possibility of a synchronization error between these EnterState calls and any operation within the StateCoder framework.
- Any public properties in your state machine object that can be accessed by both state objects and outside objects, should either be synchronized, or you should establish clear rules on accessing those properties. For example: only access those properties before the state machine's Start method is called or after it ends.
- Any public methods of your state machine that can call directly into state objects pose a risk of threading conflicts.
- Avoid exposing state objects to the outside world (allowing any direct access to state objects), or in reverse, allowing state objects access to the outside world other than through the state machine object. Doing so allows the objects exposed to be accessed simultaneously by multiple threads – which is exactly what you're trying to avoid.
- Events raised from the state machine object are always raised in the state machine's thread (with the exception of the ReachedEndState event – see next bullet). This poses a potential synchronization issue that should be considered – especially when raising events to a form or control (classes that derive from System.Windows.Forms.Control are not thread safe). Also, be

aware that if a form's thread is suspended, the attempt to raise this event will cause a deadlock.

- The ReachedEndState event automatically detects if the target of the event derives from System.Windows.Forms.Control (is a form or control) and raises the event on the correct thread. However, for all other targets the event is raised on the state machine's thread.
- The State Machine's WaitHandle is signaled before the ReachedEndState event is raised. As a rule, you should either use the WaitHandle, or the event, but not both. Trying to use both could cause a deadlock in certain situations.

As you can see, while the StateCoder framework does a great deal to protect you from threading problems when you follow the rules, it does not prevent you from creating your own sets of problems if you allow simultaneous access to objects from different threads.

Unmanaged state machines are exactly that – unmanaged

All calls into the state machine classes come from your code. These state machines do not run on the StateCoder thread pool. Therefore all threading issues and synchronization issues are up to you.

State Machines and Exceptions

One of the key design issues with StateCoder related to the handling of exceptions. Exception handling in StateCoder all draws on one basic principle: It is a very bad idea for code running in the background (in its own thread) to be able to raise exceptions that can interfere with the running of your main thread. Since these exceptions are not in the call stack for your main thread, there is no way to catch them, and allowing an application to just terminate at any time if the developer forgets to handle an exception doesn't sound like a good way to make development easier and software more stable.

For Managed State Machines

From the underlying principles just described, the following design features and development practices apply:

- There are a number of exceptions that can occur during the construction of a state machine. The most common of these is if the StateCoder framework cannot create all of the state objects defined by the state machine's ContainsState attributes. Licensing errors are also raised immediately on construction. These exceptions are raised in the code creating the state machine.
- By default, you cannot send exceptions to state machines. This is a design feature, not a limitation. The protected SendException method is intended to be called only by the StateCoder framework. You may override the SendException method to provide global exception handling for your state machine instead of handling exceptions in each state independently.
- Exceptions that your code raises during the MessageReceived and EnterState methods are immediately sent to the current state's ExceptionReceived method (internally the state machine's protected SendException method is called).

- Exceptions raised during your state's ExceptionReceived method are usually ignored. This is to avoid a stack overflow as each ExceptionReceived method raises an exception that then gets sent to the ExceptionReceived method. However, if you raise serious errors, such as attempting to set an invalid state, the state machine will abort.
- The default behavior of the state class ExceptionReceived method is to set the state machine into the end state and to set the State Machine's LastException property. This may be overridden.
- Calling the AbortStateMachine method on a state machine will almost always cause a StateAbortException to be sent to the ExceptionReceived method of the current class (internally the state machine's protected SendException method is called). It may cause this exception to be sent multiple times. However, there is a chance that it will not be called if the state machine terminates normally before the exception is processed.
- Unless you have overridden the exception handling functions, throwing an error in your state machine code causes the exception to be reflected to the ExceptionReceived method, and the state machine's LastException property is set. You can simply set the state machine's LastException property directly and use the NextState method to direct the state machine to the desired state. It does the same thing but is a bit faster (however, it also bypasses the tracing that can be performed during the state machine's SendException call).

This may sound confusing, but it is actually fairly simple. The bottom line: Unless you override the default exception handling, any exception that occurs in your state machine code will cause the state machine to go directly to the end state, and its LastException property to contain a reference to the exception.

For Unmanaged State Machines

As with threading, unmanaged state machines provide minimal protection with regards to exceptions. Specifically:

- There is no error handling for the SendMessage method. Exceptions raised during this call will be bubbled up to the caller.
- There is no error handling for the SendException method. Exceptions raised during this call will be bubbled up to the caller. To clarify: The SendException method calls the ExceptionReceived method of the current state. If the ExceptionReceived method raises an error, that exception will be raised to the caller of the SendException method.
- Exceptions that occur during the EnterState method call on a state (which is triggered by a NextState call made by a state or SetNextState call in a state machine), will be trapped and cause an immediate call to the SendException method. This means that if your EnterState implementation raises an error, you can expect the exception to immediately arrive in the form of an ExceptionReceived call.
- State machine errors (such as invalid states or licensing errors) will raise exceptions to the caller when they occur.

Design Issues for Using StateCoder with Components

There are a few additional issues to consider when using StateCoder with a component (such as a WebControl, UserControl, class library, etc.)

- If your component uses StateCoder internally to implement its functionality, and does not expose any state machines directly to the container, the container (obviously) will not need to reference the StateCoder DLL. If you need to expose an actual state machine to the container, you will need to shadow any public or protected state machine methods and events so that the container will not need to reference the StateCoder DLL.
- If a container references the StateCoder DLL directly, the container developer must own a StateCoder license.
- The ReachedEndState event will automatically synchronize to forms or controls. However, if the object receiving the event does not derive from Windows.Forms.Control, the event is raised on the state machine's thread. If you forward this event to your component's container (by raising another event), you should either warn the user of this synchronization issue, or perform your own synchronization.

State Machine Tracing and Diagnostics

As you are aware, a state machine in StateCoder is made up of a StateMachine class, and an arbitrary number of State classes (or rather, classes that derive from these base classes). As a state machine runs, it processes messages that come in to the state machine. These messages, which can take virtually any form, cause the state machine to switch from state to state. The framework can manage large numbers of state machines, handling various tasks from thread management to synchronization, dispatching messages to the appropriate place as it runs.

Traditional debugging approaches often become challenging when dealing with numerous threads and asynchronous operations, and those limitations become apparent when working with state machines as well. It can be difficult to trace the operation of a single state machine out of hundreds and to log information about the messages and state machines so that you can understand the operation and analyze any problems that may be occurring. Stepping through an application can take too long, and it can be difficult to set breakpoints because the conditions you would need may complex or unknown.

Traditional tracing techniques (such as Debug.Print in VB6) is too verbose.

To address this problem, the StateCoder framework includes built-in instrumentation based on the .NET diagnostic namespaces.

The following is a brief introduction to the .NET tracing system along with how it is extended for StateCoder. Please visit our web site for details on availability of an in-depth tutorial on .NET tracing in general.

Traditional Tracing

Every VB6 programmer has added tracing to a program using statements such as Debug.Print. In the VB6 model, such tracing is very simple as shown here:

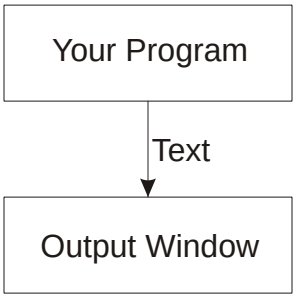


Figure 3 – Traditional “VB6” style tracing

Diagnostic output consists of simple strings that are sent to an output or debug window, and only while running in the design environment.

The diagnostic output scheme in .NET is considerably more sophisticated. It addresses a number of issues:

- Methods that allow you to generate diagnostic data consisting of arbitrary objects, not just text.
- A mechanism to specify what type of diagnostic data a program should generate. The decision on what types of data to generate can be changed using an external configuration file.
- A way to customize the processing of diagnostic data, sending it to different listeners.

Figure 4 illustrates the .NET tracing architecture.

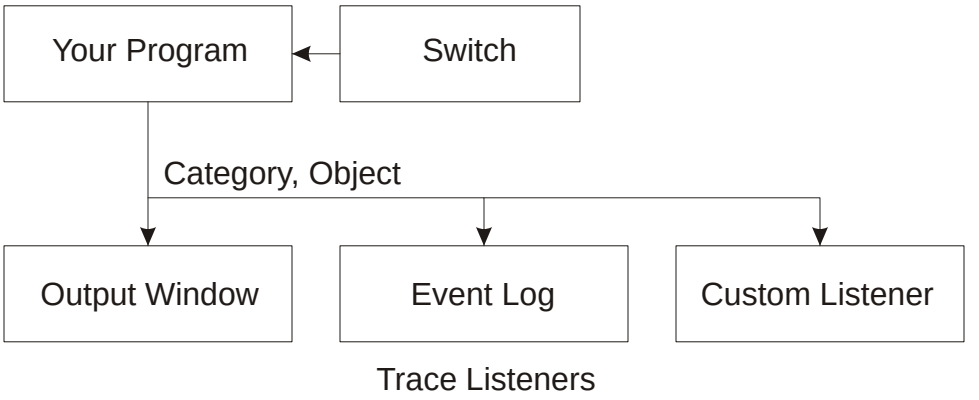


Figure 4 – Tracing in .NET

The Switch – Deciding what to report

The .NET framework comes with two Switch classes, the BooleanSwitch and TraceSwitch class. The Boolean Switch class turns tracing on or off. The TraceSwitch class allows you to specify a level from 1 to 5 indicating the severity of errors you with to examine.

Let's start by looking at the Boolean Switch.

The TracingPaper sample application creates an instance of a BooleanSwitch object when the form is loaded thus:

```
Private Shared m_BooleanSwitch As BooleanSwitch


if m_BooleanSwitch is Nothing Then m_BooleanSwitch _
= New BooleanSwitch("TraceBooleanSwitch", _
```

```
"Test of Boolean Tracing")
```

The text “TraceBooleanSwitch” identifies the name of this switch. Switches should always be Shared variables. When you create the object, the framework goes out to your application’s configuration file which has the name of your application followed by the suffix .config, and which looks something like this:

```
<configuration>
 <system.diagnostics>
 <switches>
 <add name="TraceBooleanSwitch" value="1" />
 </switches>
 </system.diagnostics>
</configuration>
```

The configuration file must be in the same directory as your executable. By changing the value of this entry, you can turn the switch on and off. In your code, you use the BooleanSwitch variable you created to decide whether or not to generate diagnostic information as shown here in the cmdBoolean button click event:

```
Private Sub cmdBoolean_Click(ByVal sender As
System.Object, _
ByVal e As System.EventArgs) Handles cmdBoolean.Click
    If m_BooleanSwitch.Enabled Then _
    Trace.WriteLine( _
    "Command1 was clicked", "Button")
End Sub
```

The TraceSwitch is similar, except that the values range from 1 to 5, and in your code instead of using the Enabled property to decide whether to output data, you check the level value of the TraceSwitch object. If the

information is more “severe” than the current level, you write it to the Trace objects.

This illustrates the general approach to tracing an application. You decide what types of switches you wish to use, and assign them names. You decide in your code what type of information to send for various switches and levels. Let me stress this – it’s up to you to decide what these switches actually mean. And you can have more than one switch active at once.

You can also define custom switches. StateCoder defines a custom switch named “StateCoderSwitch” (built of class SCTraceSwitch) that uses the following public enumeration to decide what types of information should be traced.

```
<Flags()> Public Enum TraceOptions
    Off = 0
    Undefined = 0
    EnterState = 1
    MessageReceived = 2
    StateTransition = 4
    ExceptionReceived = 8
    FinalState = 16
    Reset = 32
    All = 255
End Enum
```

You can set the SCTraceSwitch switch in your application’s configuration file. For example, the configuration file:

```
<configuration>
  <system.diagnostics>
    <switches>
      <add name="StateCoderSwitch" value="24" />
    </switches>
```

```
  </system.diagnostics>
</configuration>
```

turns on the FinalState and ExceptionReceived enumerations.

In .NET, you are not limited to tracing string. You can also pass objects to the Trace routine. StateCoder uses this internally. Instead of calling the internal Trace routines with strings, it passes them objects that contain detailed information about the event that occurred. This is the StateCoderTraceEvent object. You probably will not use this object unless you write your own custom listener.

The SCTraceListener Object

The default trace listener simply dumps a text string to the output window. StateCoder defines the SCTraceListener object to receive internal StateCoder events (all internal StateCoder trace events belong to the category “StateCoder”.

There are two ways to add listeners. You can do so from configuration files, in which case you add entries in the configuration file that specify the type of the listener and from which assembly to load it, along with initialization data. But you can also create a listener on the fly in order to add additional debugging features to your own applications. This is the usual approach when using the built in StateCoder instrumentation.

You would typically add the following code to your application’s startup code:

```
DebugListener = New
Desaware.StateCoder.SCTraceListener()
‘ Trace.Listeners.Clear()
Trace.Listeners.Add(DebugListener)
```

The reason for clearing the default listeners is that when you turn on all of the StateCoder instrumentation, you can find yourself seeing a great many events. But this is optional. As you run the program, the SCTraceListener object will be passed a reference to a StateCoderTraceEvent object each time one of the events that matches the current Switch value is detected. The information from these objects is used to load an ADO.NET DataSet object held by the listener.

In effect, this builds, on the fly, a database of detailed trace information. You can access this DataSet directly, dump it to an XML file, execute queries on it, and so on. You can also dump it into a CSV format file that can easily be read into a spreadsheet – a handy way for examining the behavior of one or more state machines.

The StateCoderAutobid and StateCoderAuctionDatabase sample projects demonstrate the use of tracing.

Licensing and Distribution

Here's the short version:

- Desaware's StateCoder is licensed on a per-machine basis. That means each computer on which you wish to develop applications using StateCoder, must have it's own license and be installed with its own unique installation code. Contact Desaware for discounted extra system licenses for use with test systems.
- There are no fees to distribute executable files, web services, Windows Services or ASP.NET applications that use StateCoder.
- You will, however, need an embedded distribution license if you wish to distribute components such as UserControls, WebControls or class libraries that use StateCoder. Each component that you wish to redistribute with StateCoder requires its own embedded distribution certificate – which you can purchase from Desaware.

Distributing an Application

To Distribute your executable (EXE, web service, Windows service or ASP.NET application):

1. Use the StateCoderCert.exe program to create a new runtime certificate for your assembly. Enter the short assembly name (not the strong name, and not the namespace) for the assembly. This will create a file named *assembly*.StateCoder.RuntimeCert.ResX.
1. Add the file *assembly*.StateCoder.RuntimeCert.ResX as a resource to your application's main assembly.
1. Distribute the file DesawareStateCoder11.dll or DesawareStateCoder20.dll with your application. You will typically install it in the same directory as your application's executables.

That's all there is to it!

Distributing a Component

To Distribute your component (UserControl, WebControl or class library):

3. Contact Desaware to purchase an embedded license for your component. You will be provided with an embedded installation code.
3. Use the StateCoder Embedded Certificate Utility to create an embedded certificate for your component. Enter your installation code and the name of the assembly. This will create a file named *assemblyname*.StateCoder.EmbeddedCert.Resx. This file is your embedded certificate.
3. Add the embedded certificate as a resource to your components main assembly.
3. Distribute the file StateCoder.dll with your application. You will typically install it in the same directory as your application's executables.

This will allow others to use your component in design mode and to debug it, even though they do not have the StateCoder product.

WARNING!

Do not use the same installation code on more than one machine when installing StateCoder.

Do not use the same embedded installation code to create certificates for more than one component.

We encourage you to read about specific design issues relating using StateCoder with components.

More On Licensing

When designing our licensing system, we wanted to set a balance. On one hand, we wanted to provide reasonable protection for our software. On the other hand, we really dislike very long installation codes, internet based activation codes, and especially licensing schemes that could cause some outsider to disable your applications.

So we came up with this certificate based scheme that we believe will be a fair compromise. The general idea is as follows:

Demo mode

The StateCoder component that you download from our site as a demo, is the actual StateCoder component. However, without the product installed, the component runs in demo mode. This means it will only work with certain assemblies - specifically, the ones we provide as demonstration versions and any assembly named StateCoderDemo.

This allows you full functionality of the product for evaluation purposes, but is obviously not suitable (or licensed) for use in your own applications or components, or for further distribution.

Design/Debug Mode

Once you install the StateCoder product, the StateCoder component is enabled for use on that development system.

If you copy applications that use this component onto other development systems (which is common in team development environments) or onto test systems, everything will work fine as long as each system has a unique installation code.

However, if you have used the same installation code on more than one system (which of course you wouldn't do because it is a violation of your license), the component will not work.

Runtime Distribution with Applications

When A StateCoder component lands on a system that does not have StateCoder installed, it by default returns to demo mode – which isn't particularly useful in terms of allowing your application to run.

When you installed StateCoder, the installer created a runtime certificate file named StateCoder.RuntimeCert.resx. When you add this to your top level assembly, it informs StateCoder that it should not enter demo mode, but rather should run normally.

Embedded Distribution with Components

A runtime certificate will allow an application to run, but will not permit debugging of the application. When distributing components, you obviously will want your clients to be able to debug their applications that use your components.

To allow this you may purchase an embedded license. The embedded installation code you purchase from Desaware will allow you to create an embedded certificate that is bound to your component's assembly name. This will be a file named *yourassembly*.StateCoder.EmbeddedCert.ResX. When you add this to the top level assembly of your component, it informs StateCoder that it should be allow debugging.

But remember – don't use the embedded installation code to create more than one certificate. If you do so, and StateCoder detects it, it will cause a license violation.

Switching between Computers

Each StateCoder installation is bound to a computer based on the computer name. It is our experience that developers rarely change the name of their computer, so this seems a reasonable approach. If you wish to move the product from one machine to another (i.e., uninstall from one

machine, and install on another using the same installation code), you must do the following:

- Uninstall StateCoder from the first machine.
- Install StateCoder on the new machine.
- Recreate any embedded certificates for your components that were created on this machine (you may use the same embedded installation codes as before).
- Rebuild any applications or components that were built on this system using the new certificates.

Please keep in mind that Desaware has very reasonably priced multiple unit licenses. Also, if you have an unusual scenario, please call us and we'll work things out.

Technical Support

For information on customer support and last minute changes, refer to the file readme.wri on the StateCoder CD (or in the main application directory for electronic downloads). This file is compatible with write.exe (included with each copy of Windows).

There is a saying in the software world that no non-trivial program is completely bug free. The corollary to that saying is that no program with more than 10 lines in it is non-trivial. StateCoder is emphatically non-trivial....

StateCoder has undergone extensive testing to make it as bug free as possible. Nevertheless, it is possible that some have crept through. Please write or send us a fax if you find one, and include all of the steps needed to reproduce the problem. Also, if there are any files needed to reproduce the error, send them to us via Email.

StateCoder is a class framework. While we have done, and will continue to do everything possible to ensure that the framework is bug free, it is not possible for us to provide general support on .NET, or on specific applications using StateCoder. In other words – we cannot debug your code for you. We strongly encourage you to read the documentation carefully and make sure that your code follows the guidelines provided.

If you have any questions you are also welcome to refer to our Frequently Asked Questions section of our web site.

We would also appreciate your suggestions regarding this manual. Specific comments and questions are especially welcome. We have attempted to address as many questions as possible, but if you run into something confusing, please let us know so that we can incorporate revisions into the next edition and post them to our web site.

Finally, and perhaps most important, we would love to hear your suggestions for improvements to StateCoder, or any suggestions you may have for new products or components.

Please address all correspondence to:

Desaware, Inc.
3510 Charter Park Drive, Suite 48

San Jose, CA 95125

Telephone: 408/404-4760 Fax: 408/404-4780

Web Site: http://www.desaware.com

E-mail: support@desaware.com