Working With
North State Framework in C#
V2.0

North State Software, LLC
www.northstatesoftware.com

Contents

1 Overview

The Unified Modeling Language™ (UML™) specifies a set of diagrams for describing object-oriented software. These drawings facilitate design and development by communicating software architecture at a high level of abstraction using a standards-based description language.

UML 2.3 includes seven diagrams for describing the static (time-invariant) structure of a system, and seven diagrams for describing dynamic behavior. Many of these diagrams, particularly the static structure diagrams, are easily translated in source code. However, implementing dynamic behaviors, such as those described by UML State Machines, poses a more difficult problem.

UML state machines, based on Harel statecharts, are a powerful and compact notation for describing behavior. Simple state machines can be coded with "switch" statements or nested if-then-else constructs; however, these implementations are often tedious and error prone, particularly as the state machine grows in size. In addition, such implementations are difficult to extend or reuse through generalization.

The North State Framework™ (NSF™) simplifies the process of implementing UML state machines. The framework classes map directly to UML diagram elements, so that working executable code can be created in a straightforward, methodical fashion. The North State Framework is perfect for hand-coding UML model based designs or embedding into UML based modeling tools.

In short, the North State Framework bridges the gap between UML and code, enabling you to build better solutions, faster.

1.1 Advantages

The North State Framework simplifies the process of turning UML state machines into executable code. The one-to-one relationship between diagram elements and framework classes makes it easy to understand and implement.

Designed with extensibility in mind, the North State Framework allows extension through both inheritance and composition. Inherited state machines can add, remove, or change behavior from their bases class. In addition, state machines are "pluggable", so that one state machine can be used inside another state machine, thus promoting reuse through composition.

UML model based development facilitates design at a high level of abstraction, thus speeding development, improving collaboration, and minimizing design errors. There are several UML based development environments available, but many companies don't want to be tied to their code generation tool or can't afford their price tag. The North State Framework provides an alternative. It allows developers their choice of modeling tools, from hand sketches to full-blown modeling environments, and provides the classes to quickly and reliably implement their designs.

For UML tool developers, the North State Framework provides an entry point into modeling dynamic behaviors. Many UML based modeling tools do not support code generation from state machines, and if they do, often the generated code is neither extensible nor suitable for production quality software, failing to adequately handle threading concerns or exception handling. With the North State Framework, extensibility, threading, and exception handling is built into the framework, reducing development efforts and improving product quality. And because the North State Framework classes directly map to state machine diagram elements, it is easily integrated into any diagram to code generation engine.

1.2 Features

The North State Framework supports the following features:

- State Machines – semantically correct, fully functional, and extensible
- States – including initial, choice, composite, deep history, and shallow history
- Delegates – dynamically add or remove state entry, state exit, or transitions actions
- Events – simple and payload carrying
- Transitions – including internal, local, and external
- Regions – implement concurrent behaviors
- Fork/Join – synchronize current behaviors
- Threads – assign one or more state machines to run on a thread
- Timers – schedule an event or execution of an action
- Inheritance – easily extend base state machine behavior
- Composition – plug one state machine into another
- Trace Logging – record state machine history

1.3 Scope and References

This document contains best practices for working with the North State Framework. It assumes the reader is somewhat familiar with UML State Machine notation and semantics. If this is not the case, there are numerous references available on the web, including the UML specification, which can be downloaded from www.omg.org.

2 Best Practices Using the North State Framework

Following are recommended best practices when using the North State Framework. These conventions have resulted from our years of experience designing and implementing state machine based solutions for complex systems. If we could sum up these practices into one sentence, it would be:

Design and code your state machines to be extensible.

The framework is designed to help achieve this goal, allowing your design to be flexible, and enabling creation of powerful and reusable components.

2.1 Development Process

Most developers benefit from laying out their designs graphically before diving into the details of the code. UML state machines are designed for just this purpose, enabling you to design graphically at a high-level, without worrying about all the underlying code details. Once your state machine design is complete, NSF will enable you to quickly and easily translate your UML State Machines into error-free, executable code.

Here are some things to remember when designing state machines:

- A state machine can represent the behavior of a single object or a process coordinating several objects.
- State machines can be used to monitor or control objects.
- Each state in the state machine should represent a unique condition that exists for some significant period of time.
- Be careful not to use state machines as flowcharts. Un-triggered, un-guarded transitions are a tell-tale sign that you have used a state machine as a flowchart. These result in states that don't exist for significant periods of time.

Developing with NSF allows for a methodical process to create executable code from UML State Machines, as described below.

1. Draw state machine
2. Create the state machine class
3. Add state machine members
4. Write constructor
5. Write actions
6. Write guards
7. Run it!

2.2 Draw the State Machine

The act of drawing out the state machine will help you identify holes in your behavioral logic, before getting to the intricacies of coding. Don't be fooled by the simplicity of this step. The time spent on a well designed state machine may exceed the time for the implementation, and be well worth it. Do a good job here, and you'll save yourself time and frustration. For the example used herein, we used Microsoft Office Visio™ to draw our UML state machine, as shown in Figure 2-1.

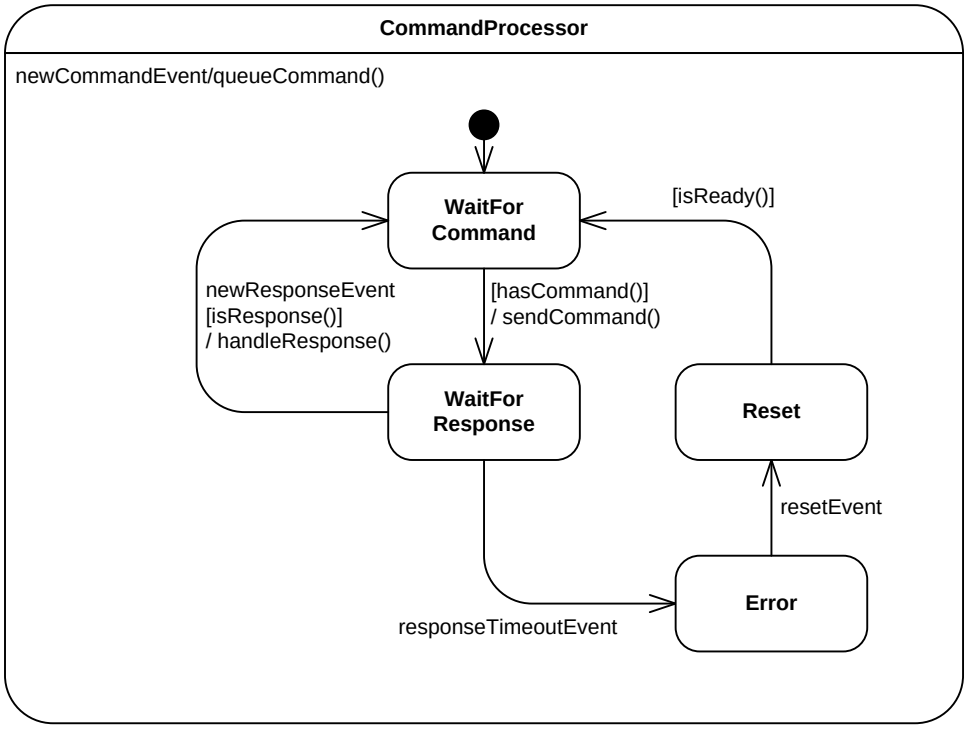

Figure 2-1: Command Processor Example

As you design your state machine, keep in mind that there are five places for actions (behaviors) to occur:

1. On state entry.
2. On state exit.
3. On transition from one state to another.
4. As a reaction in state to an event (internal transition).
5. As a timer action.

Tips:

- Start with the "Happy Path" scenario on the first pass. Then add error handling details as the design matures.
- Name diagram members appropriately, including states, events, triggers, guards. Keep in mind that software Q/A or service personnel from your company may need to understand the state machine design from the diagram.
- Avoid putting implementation details on the drawing. Instead, rely on good naming at the diagram level, and leave the implementation details for the code. This will help you avoid diagram changes when implementation details change.
- Think the implementation. Plan what actions need to occur on the entry to, exit from, transitions between, and as reactions within each state. Annotate the drawing with notes describing what actions should occur, without specifying programmatically how to implement them.
- Look for race conditions in your logic. Many state machine designs are multithreaded, so pay careful attention to threading issues.
- Use the diagram to expose weaknesses in the design. A convoluted state machine diagram is an indication of a convoluted design.

2.3 Create the State Machine Class

The first step in transforming your state machine into working code is to create a class that inherits from NSFStateMachine, as illustrated below.

```
public class CommandProcessor : NSFStateMachine
{
  …
}
```

2.4 Add State Machine Members

The second step in transforming your state machine into working code is to add members for each state machine element.

Although it is not always necessary to create members for every state machine element, it is recommended practice to do so. These members will be useful in the event that you want to extend the state machine, either via inheritance or composition. Another reason they are helpful is for debugging, so that you can examine the properties and relationships amongst members.

NSF is designed such that UML State Machine elements have corresponding NSF classes, which makes translation very straightforward. NSF contains the following classes to implement UML State Machines:

- NSFStateMachine – A state machine.
- NSFRegion – An area containing nested states and transitions.
- NSFState – A simple state that cannot contain nested regions.
- NSFCompositeState – A state that can contain nested states, regions, or submachines.
- NSFInitialState – The default entry point for a region.
- NSFChoiceState – A state that is used for decision branching.
- NSFDeepHistory – A state that allows returning to a prior state and all its substates.
- NSFShallowHistory – A state that allows returning to a prior state, but not its substates.
- NSFForkJoin – A state that provides synchronization and branching across multiple regions.
- NSFEvent – A trigger mechanism for transitions and reactions.
- NSFDataEvent – An NSFEvent that carries a data payload.
- NSFExternalTransition – A transition path from one state to another.
- NSFLocalTransition – A transition path from one state to another, where the source state is not exited.
- NSFInternalTransition – A state's internal response to a trigger event. Also known as a reaction in state.
- NSFForkJoinTransition – A specialized NSFExternalTransition that can be used for transitioning between fork-joins.

The recommended translation process is:

1. Declare event members for each unique event.
2. Declare region and state members, from outer to inner
3. Declare transition members.

Grouping states and transition together with their parent region or composite state helps organize the state machine members.

Following this process for the Command Processor Example in Figure 2-1 yields:

```
public class CommandProcessor : NSFStateMachine
{
   …
   // Events
   protected NSFDataEvent<String> newCommandEvent;
   protected NSFDataEvent<String> newResponseEvent;
   protected NSFEvent responseTimeoutEvent;
   protected NSFEvent resetEvent;

   // Regions and states, from outer to inner
   protected NSFInitialState initialCommandProcessorState;
   protected NSFCompositeState waitForCommandState;
   protected NSFCompositeState waitForResponseState;
   protected NSFCompositeState errorState;
   protected NSFCompositeState resetState;

   // Transitions, ordered internal, local, external
   protected NSFInternalTransition reactionToNewCommand;
   protected NSFExternalTransition initialCommandProcessorToWaitForCommandTransition;
   protected NSFExternalTransition waitForCommandToWaitForResponseTransition;
   protected NSFExternalTransition waitForResponseToWaitForCommandTransition;
   protected NSFExternalTransition waitForResponseToErrorTransition;
   protected NSFExternalTransition errorToResetTransition;
   protected NSFExternalTransition resetToWaitForCommandTransition;
   …
}
```

Tips:

- Create an initial state within each composite state and region whenever there are multiple substates. Although not always necessary, this approach helps add clarity to the diagram and avoids the mistake of forgetting to specify an initial state.
- Make state members of type NSFCompositeState, rather than NSFState, in case you want to extend the state by adding substates to it. NSFCompositeState provides capabilities for nesting states, regions, and submachines.
- Count the number of graphical elements on the state machine diagram and make sure it matches the number of state machine members in the code.

2.5 Write a Constructor

Once the state machine members are declared, the next step is to define them in a constructor. Each NSF class has several constructors which allow the objects to be tagged with names, as well as supply the requisite information for creation. Names are most useful for viewing in the automatic trace logging facilities.

The constructor for the Command Processor Example in Figure 2-1 is shown below.

```
public CommandProcessor(String name)
   : base(name, new NSFEventThread(name))
{
   // Events
   newCommandEvent = new NSFDataEvent<String>("NewCommand", this, "CommandPayload");
   newResponseEvent = new NSFDataEvent<String>("NewResponse", this, "ResponsePayload");
   responseTimeoutEvent = new NSFEvent("ResponseTimeout", this);
   resetEvent = new NSFEvent("Reset", this);

   // Regions and states, from outer to inner
   initialCommandProcessorState = new NSFInitialState("InitialCommandProcessor", this);
   waitForCommandState = new NSFCompositeState("WaitForCommand", this, null, null);
   waitForResponseState = new NSFCompositeState("WaitForResponse", this,
     waitForResponseEntryActions, waitForResponseExitActions);
   errorState = new NSFCompositeState("Error", this, errorEntryActions, null);
   resetState = new NSFCompositeState("Reset", this, resetEntryActions, null);

   // Transitions, ordered internal, local, external
   reactionToNewCommand = new NSFInternalTransition("ReactionToNewCommand", this,
     newCommandEvent, null, queueCommand);
   initialCommandProcessorToWaitForCommandTransition =
     new NSFExternalTransition("InitialToWaitForCommand", initialCommandProcessorState,
     waitForCommandState, null, null, null);
   waitForCommandToWaitForResponseTransition =
     new NSFExternalTransition("WaitForCommandToWaitForResponse", waitForCommandState,
     waitForResponseState, null, hasCommand, sendCommand);
   waitForResponseToWaitForCommandTransition =
     new NSFExternalTransition("WaitForResponseToWaitForCommand", waitForResponseState,
     waitForCommandState, newResponseEvent, isResponse, handleResponse);
   waitForResponseToErrorTransition = new NSFExternalTransition("WaitForResponseToError",
     waitForResponseState, errorState, responseTimeoutEvent, null, null);
   errorToResetTransition = new NSFExternalTransition("ErrorToReset", errorState,
     resetState, resetEvent, null, null);
   resetToWaitForCommandTransition = new NSFExternalTransition("ResetToWaitForCommand",
     resetState, waitForCommandState, null, isReady, null);
}
```

Tips:

- Notice that state entry and exit actions, as well as transition guards and actions can be specified via the NSF class constructors. This approach is discussed in more detail in the next section.
- Defining the members in the prescribed order will ensure that relationships are properly established.
- Supply names for all the states and events to facilitate tracing. Regions and transitions may not need names, but they can be useful in exceptional cases.

2.6 Write Actions

State machine actions are delegate methods that can be executed on state entry, state exit, and during transitions. Most of your state machine development time will be spent designing the state machine diagram and writing the underlying actions.

For the Command Processor Example, actions are associated with their state machine element during member construction, as illustrated below.

```
    waitForResponseState = new NSFCompositeState("WaitForResponse", this,
      waitForResponseEntryActions, waitForResponseExitActions);
```

Following are some of the action implementations for this example.

```
private void queueCommand(NSFStateMachineContext context)
{
   commandQueue.Enqueue(((NSFDataEvent<string>)(context.Trigger)).Data);
}

private void sendCommand(NSFStateMachineContext context)
{
   string commandString = commandQueue.Peek();

   // Code to send the command goes here
   // …
}

private void waitForResponseEntryActions(NSFStateMachineContext context)
{
   // Schedule timeout event, in case no response received
   responseTimeoutEvent.schedule(responseTimeout, 0);
}

private void waitForResponseExitActions(NSFStateMachineContext context)
{
   // Unschedule the timeout event
   responseTimeoutEvent.unschedule();

   commandQueue.Dequeue();
}
```

```
private void handleResponse(NSFStateMachineContext context)
{
   // Code to handle the response goes here
   // …
}

private void errorEntryActions(NSFStateMachineContext context)
{
   // Code to handle the error goes here
   // …
}

private void resetEntryActions(NSFStateMachineContext context)
{
   // Code to reset hardware goes here
   // …
}
```

In addition to specifying actions during member initialization, they can be added or removed on the fly with the += and −= operators, as illustrated below.

```
waitForResponseState.EntryActions += waitForResponseEntryActions;

waitForResponseState.EntryActions -= waitForResponseEntryActions;
```

This feature creates a powerful run-time facility to observe and react to state machine change. For example, if a state machine exposes one of its underlying states, a client object can connect a state entry action to that state, thus registering a delegate for notification whenever the state is entered.

The method signature for a state machine action is:

```
void methodName(NSFStateMachineContext context);
```

The NSFStateMachineContext argument provides contextual information for the action, such as the entering state, exiting state, triggering event, and transition taken.

Tips:

- Name the action method after the associated state or transition. For example, the entry actions for the error state are defined in the method "errorEntryActions".
- Make use of the action delegate feature to add and remove entry/exit actions from state machines at any time. A common pattern is a state machine observer. When adding state machine observers during run-time, be sure to remove them at the appropriate time.
- Carefully consider race conditions when monitoring or observing state machine execution. A common mistake is to check if a state is active, and if not, then add an entry action to receive notification that it has become active. This creates a race condition because the state could change between the time it is checked and the time the action is added. The preferred pattern is to register the entry action, then check the state, then unregister the action when the state is active.

2.7 Write Guards

Guards are boolean delegates that must evaluate true for transitions to execute. They are implemented in much the same way as actions.

For this example, guards are specified during member construction, as illustrated below.

```
waitForCommandToWaitForResponseTransition =
     new NSFExternalTransition("WaitForCommandToWaitForResponse", waitForCommandState,
     waitForResponseState, null, hasCommand, sendCommand);
```

The guard implementations for the example are illustrated below.

```
private bool hasCommand(NSFStateMachineContext context)
{
   return (commandQueue.Count != 0);
}

private bool isResponse(NSFStateMachineContext context)
{
   // Code to verify that the resposne is correct for the command goes here.
   // …

   return true;
}

private bool isReady(NSFStateMachineContext context)
{
   // Code to check if hardware is reset properly goes here
   // …

   return true;
}
```

Just like actions, guards can be added or removed on the fly with the += and −= operators, although this is much less common in practice. Typically, guards would only be added or removed in derived classes that change the conditions under which a transition can be taken. Outside of this use case, changing guards on the fly should be carefully scrutinized, as it could result in non-obvious behavior.

Transition guards are lists of delegate methods with the following signature:

```
bool methodName(NSFStateMachineContext context);
```

Tips:

- Avoid putting side effects in guard methods, because they may be evaluated multiple times before a transition is taken.
- Carefully consider the design before adding and removing guards on the fly.

2.8 Run It!

To run your state machine, create an instance and call the method startStateMachine(), as illustrated below.

```
static void Main(string[] args)
{
   CommandProcessor commandProcessor = new CommandProcessor("CommandProcessor");
   commandProcessor.startRunning();
   …
}
```

The framework handles all the execution details, ensuring compliance with UML semantics, including run-to-completion.

3 Features and Advanced Techniques

This section describes the features of NSF in more detail, and illustrates some advanced techniques for constructing state machines.

3.1 Threading

State machines can be run collectively on a common thread or individually on their own thread. The NSFEventThread class is provided to simplify state machine thread management. The thread is typically specified during construction, but can be reassigned at any time. In the Command Processor Example, a new thread is created during construction, as illustrated below.

```
public CommandProcessor(String name)
   : base(name, new NSFEventThread(name))
{
   …
}
```

The NSFStateMachine class does not manage creation of NSFEventThreads, rather, your application will manage the threading model. To run a state machine on a run-time specified thread, create a constructor which allows specification of the thread, as illustrated below.

```
public CommandProcessor(String name, NSFEventThread thread)
  : base(name, thread), …
```

The state machine's thread is available via the EventThread property.

Because NSF applications are often highly multi-threaded, it is important to consider mutual exclusion for any resource that may be accessed from multiple threads. The following example illustrates how to effectively lock a shared resource.

```
lock(someMutex)
{
   // Access some shared resource
   …
}
```

Tips:

- Spend some time thinking about the threading model for the application. Generally, several state machines can share a common thread, which will help minimize context switching and use of system resources. Often times, state machines within a common subsystem can share a thread, while high-priority or device driver state machines require a dedicated thread.
- Set up each state machine's thread before starting the state machine.

- Avoid blocking calls within a locked region of code. Within the framework, all locked regions of code are designed to complete quickly once the synchronizing object is obtained, thus eliminating deadlock situations.

3.2 Accessors

State machines can easily be extended through inheritance, composition, or a combination of both. Part of the design process involves deciding which parts of the state machine to expose to derived or composite state machines, and even for classes that interact with the state machine. Accessors are a good way to expose those part of the state machine with which other classes may interact.

For the CommandProcessor example, the following accessors would be useful:

```
//States
public NSFCompositeState WaitForCommandState { get { return waitForCommandState; } }
public NSFCompositeState WaitForResponseState { get { return waitForResponseState; } }
public NSFCompositeState ErrorState { get { return errorState; } }
public NSFCompositeState ResetState { get { return resetState; } }

// Transitions
public NSFExternalTransition WaitForCommandToWaitForResponseTransition
  { get { return waitForCommandToWaitForResponseTransition; } }
public NSFExternalTransition WaitForResponseToWaitForCommandTransition
  { get { return waitForResponseToWaitForCommandTransition; } }
public NSFExternalTransition WaitForResponseToErrorTransition
  { get { return waitForResponseToErrorTransition; } }
public NSFExternalTransition ErrorToResetTransition
  { get { return errorToResetTransition; } }
public NSFExternalTransition ResetToWaitForCommandTransition
  { get { return resetToWaitForCommandTransition; } }
```

Tips:

- Create accessors for states and transitions to allow other classes to:
    - Query if a state is active
    - Add and remove actions

3.3 Events

Events are used to trigger state transitions. This process starts when an event is queued to the state machine's thread through the queueEvent(…) methods, available in both the state machine and event interfaces. Events in the thread's event queue are processed one at a time in the order they were queued. The processing sequence for an event is that it is first sent to the most deeply nested active state in the state machine (concurrent behavior is slightly different; see the section on concurrency for more detail). If the event causes a transition, then the event is considered to be "handled" and processing of that event stops. If the event is "unhandled", then it is sent to the parent state where the same logic is applied. It is possible that an event will pass all the way up to the top level state machine and still be unhandled, whereupon the event will just be discarded. However, if the event is handled anywhere along the chain, processing of that event stops, at which time the state machine will then evaluate if any more transitions should occur, based on the new state of the state machine (following UML run-to-completion semantics). The figure below illustrates an event handling scenario. See the UML specification for more detail on state machine event handling.

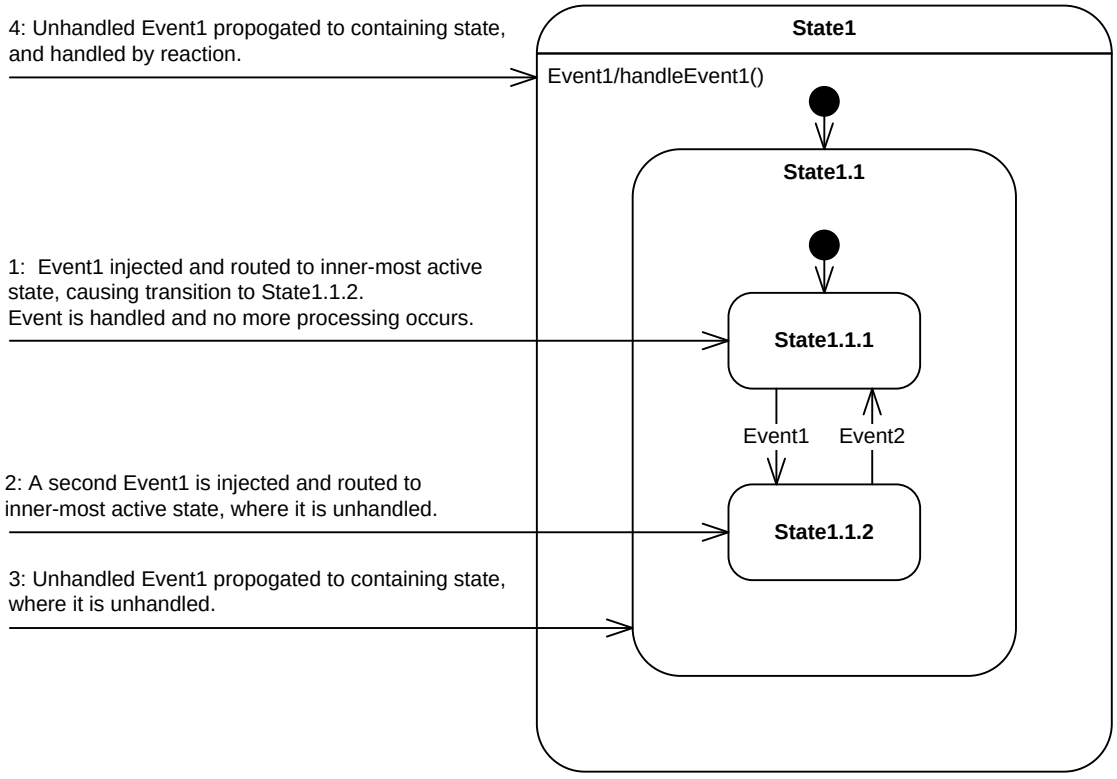

Figure 3-1: Event Propagation

NSF events contain a unique identifier that is used to compare the event in process to the event specified as part of transition construction. If the identifiers match, then the event satisfies the trigger requirement. Of course, if the event queued to the state machine's thread is the same event used to define the transition or reaction trigger, then the identifiers will match. However, there are cases when it is beneficial to have multiple unique events that can trigger the same transition. The most common case is when each event carries specific payload data (see NSFDataEvent in the NSF documentation). To handle this situation, use the copy(…) method to create copies of the original event, so that each copy will have the same unique identifier. In the case of NSFDataEvent objects, each copy can contain a unique data payload, but will still share the same unique identifier if the copy(…) method is used. This technique is illustrated below.

```
public class CommandProcessor : NSFStateMachine
{
   public CommandProcessor(String name)
      : base(name, new NSFEventThread(name))
   {
      …
      newCommandEvent = new NSFDataEvent<String>("NewCommand", this);
      …
      reactionToNewCommand = new NSFInternalTransition( … , newCommandEvent, … );
      …
   }
   …
   protected NSFDataEvent<String> newCommandEvent;
   …
   protected NSFInternalTransition reactionToNewCommand;
   …
   void addCommand(String newCommand)
   {
      // The following event will have the same identifier as the newCommandEvent.
      // It can be used interchangeably to trigger the reactionToNewCommand.
      queueEvent(newCommandEvent.copy(newCommand));
   }
   …
}
```

Tips:

- Rather than exposing state machine events as part of the public interface, consider writing methods which internally queue the approprate events. This approach can make the client interface more easily understood and allows for internal changes in the future.

3.4 Timers

The NSFEvent class contains methods that make it easy to schedule an event to be queued at a specified time, as illustrated below.

```
…
NSFEvent timeoutEvent;
…
void someStateEntryActions(NSFStateMachineContext context)
{
  // Automatically timeout after 1 second
  timeoutEvent.schedule(1000, 0);
}

void someStateExitActions(NSFStateMachineContext context)
{
  // Unschedule timeout in case leaving due to another event
  timeoutEvent.unschedule();
}
```

A common pattern for a timeout event is to schedule it in the state's entry actions and to unschedule it in the state's exit actions. This ensures that the event won't fire if the state is exited for some reason other than the timeout event. If the state is exiting due to the timeout event, the unschedule call has no effect.

Behind the scenes, the NSFEvent class uses the primary timer within the NSFTimerThread class to implement the schedule and unschedule behaviors. This timer is setup to run with a resolution of 1 mS (hardware must support), and can be used to schedule different types of actions.

Three types of timer actions are natively supported, events, scheduled actions, and signals. The NSFScheduledAction class allows an action (delegate) to be called at a specified time on a specified thread. See the online documentation for more details on these classes.
Tips:

- The primary timer in NSFTimerThread class can be accessed with the NSFTimerThread.PrimaryTimerThread property.
- The primary timer runs on its own thread at the highest thread priority.

3.5 Pseudo States

Non-persistent states are generically referred to as "pseudo states". These include initial states, choice states, deep history states, and shallow history states. Because they are non-persistent, outgoing transitions must not contain triggers and must be guarded such that the pseudo state exits immediately after entering. Correspondingly, internal and local transitions are not valid for pseudo states.

3.5.1 Initial States

An initial state indicates the default entry point for a set of nested states in a composite state. Initial states must have a default (un-triggered and un-guarded) outgoing transition. The symbol for an initial state with a default transition is:

Figure 3-2: Initial State with Default Transition

Although not always necessary, it is good practice to include an initial state whenever a composite state or region contains multiple nested substates. This approach helps add clarity to the diagram and avoids the mistake of forgetting to specify an initial state.

3.5.2 Choice States

A common state machine pattern is to exit a state based on an event and transition to one of several states based on a guard. The choice state helps implement this pattern, as illustrated below:

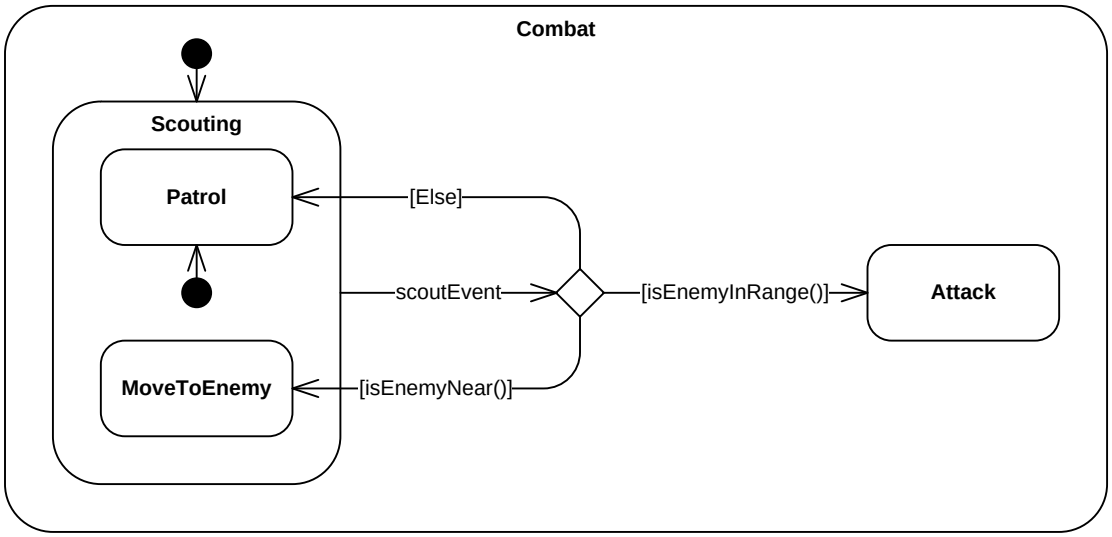

Figure 3-3: Combat Example

Choice states have multiple outgoing transitions, each with its own unique guard, and should have one outgoing default transition labeled "Else" which is taken if all other transition guards fail.

The NSFChoiceState class implements the state machine choice pseudo state. It handles the logic of identifying the default "Else" transition and makes sure it is taken only if all other transition guards fail. A portion of the class declaration for the Combat Example state machine is shown below.

```
public class Combat : NSFStateMachine
{
   …
   // Events
   protected NSFEvent scoutEvent;

   // Regions and states, from outer to inner
   protected NSFInitialState combatInitialState;
   protected NSFCompositeState scoutingState;
   protected NSFInitialState scoutingInitialState;
   protected NSFCompositeState patrolState;
   protected NSFCompositeState moveToEnemyState;
   protected NSFChoiceState attackChoiceState;
   protected NSFCompositeState attackState;

   // Transitions, ordered internal, local, external
   protected NSFExternalTransition combatInitialToScoutingTransition;
   protected NSFExternalTransition scoutingToAttackChoiceTransition;
   protected NSFExternalTransition scoutingInitialToPatrolTransition;
   protected NSFExternalTransition attackChoiceToPatrolTransition;
   protected NSFExternalTransition attackChoiceToMoveToEnemyTransition;
   protected NSFExternalTransition attackChoiceToAttackTransition;
   …
   private bool isEnemyNear(NSFStateMachineContext context);
   private bool isEnemyInRange(NSFStateMachineContext context);
};
```

Tips:

- Transitions without triggers entering into choice states are a warning sign that the state machine is being used as a flowchart. Don't fall into this trap. Make sure states represent persistent conditions.
- Don't put triggers on transitions out of choice states.
- Watch for race conditions when evaluating the guards for transitions emanating from choice states.

3.5.3 History States

History states allow a composite state to return to the last active condition before the state exited. There are two types of history states, deep history and shallow history. A deep history will restore the last condition of the composite state and all of its nested composite states. A shallow history will restore the last condition of the composite state, but will not restore any further nested composite states.

The following diagram illustrates the use of a history pseudo state. Notice that the history state can have one outgoing null transition to a default state. This transition is taken only on the first entry into the history state. All subsequent entries will return the composite state to the last active substate, and in the case of a deep history, all further nested substates.

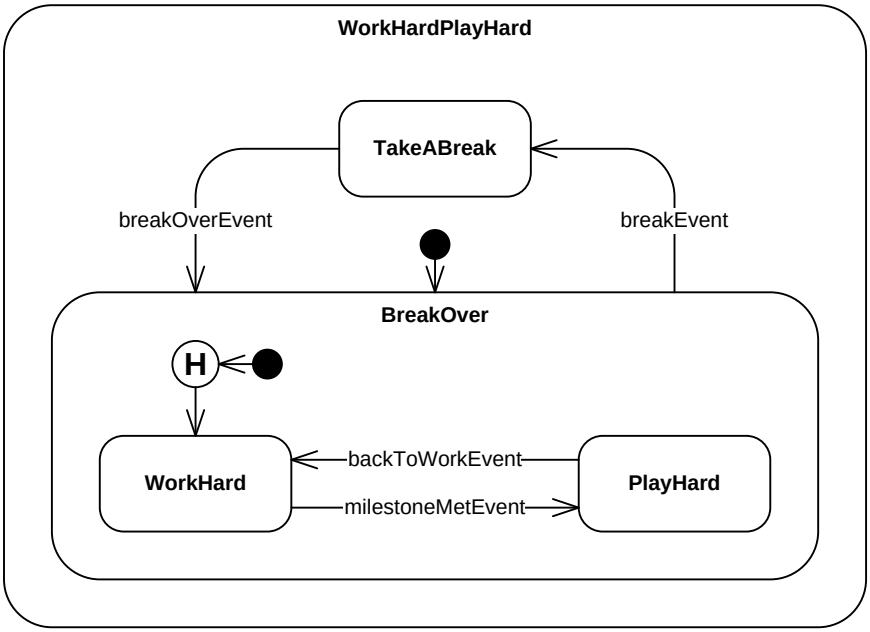

Figure 3-4: Work Hard Play Hard Example

The source code for this state machine is included in the NSF examples.

Tips:

- Be sure to provide a default transition out of the history pseudo state to indicate what should happen on first entry.

3.6 Composite States and Concurrency

As seen in the examples above, nested substates can be easily implemented by specifying the parent composite state in their constructor. Alternatively, substates can be implemented by first creating a region within a composite state, and then nesting substates within the region. Another alternative, illustrated in later chapters, is to nest submachines within a composite state.

Concurrent behavior is implemented in a state machine by creating multiple regions within a composite state, each region having its own set of substates. When an event is processed by a composite state, all regions within the substate are given the opportunity to handle the event, even if the event was already handled by another substate in a concurrent region. Remember that an event is considered "handled" only if it causes a transition to occur. Otherwise, the event is considered "unhandled". Concurrent behavior is different than non-concurrent behavior, wherein the first state to handle the event is the only state that will have an opportunity to handle the event.

Tips:

- It is easy to get carried away with nested substates and concurrent regions. Don't try to fit the behavior of an entire system into one state machine. The most flexible state machine designs (and software designs in general) group only those behaviors that truly require coupling. Remember, a key philosophy in good software design is to "divide and conquer".

Regions are often organized into "swim-lanes" as illustrated in Figure 3-5 below. In this example, a hypothetical system contains two subsystems that must be coordinated by the system. This example also illustrates the use of a fork-join to provide synchronization and branching across regions. A fork-join provides synchronization by requiring all incoming transitions to occur before the outgoing transitions occur. Once all incoming transitions have taken place, all outgoing transitions will be taken. It is not valid to place a trigger or a guard on a transition exiting a fork-join. Fork-joins are the only mechanism that can affect a transition that crosses region boundaries, because it is not valid to have a transition between states in different regions.

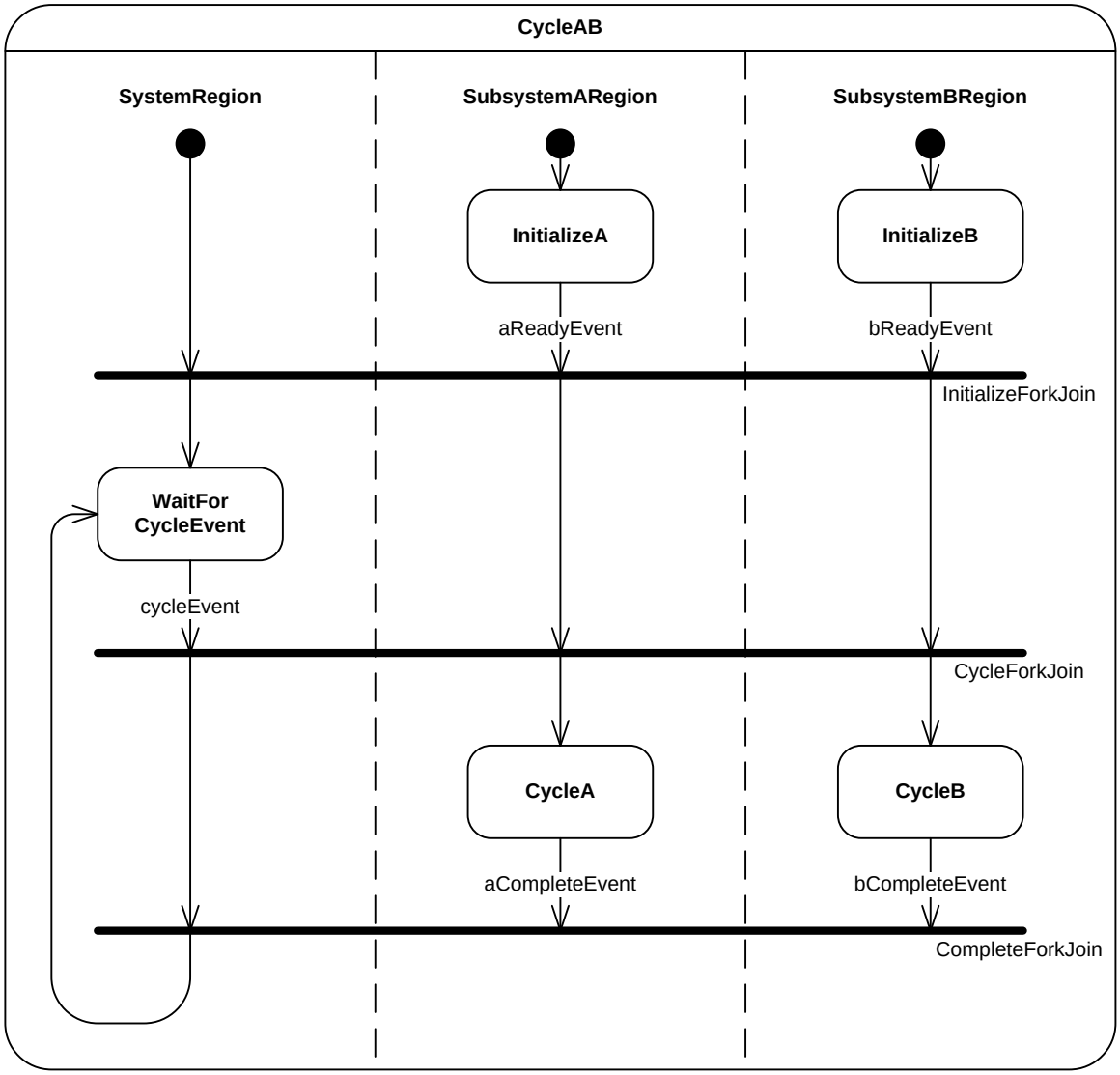

Figure 3-5: CycleAB Example

As with all NSF classes, fork-joins are easy to implement because there is a one-to-one correlation between graphical elements and NSF classes. The following source code illustrates part of the class declaration for the CycleAB Example state machine.

```
public class CycleAB : NSFStateMachine
{
   …
   // Events
   protected NSFEvent cycleEvent;
   protected NSFEvent aReadyEvent;
   protected NSFEvent bReadyEvent;
   protected NSFEvent aCompleteEvent;
   protected NSFEvent bCompleteEvent;

   // Regions and states, from outer to inner
   protected NSFRegion systemRegion;
   protected NSFRegion subsystemARegion;
   protected NSFRegion subsystemBRegion;
   protected NSFForkJoin initializeForkJoin;
   protected NSFForkJoin cycleForkJoin;
   protected NSFForkJoin completeForkJoin;

   // System Region
   // Regions and states, from outer to inner
   protected NSFInitialState systemInitialState;
   protected NSFCompositeState waitForCycleEventState;
   // Transitions, ordered internal, local, external
   protected NSFExternalTransition systemInitialToInitializeForkJoinTransition;
   protected NSFExternalTransition initializeForkJoinToWaitForCycleEventTransition;
   protected NSFExternalTransition waitForCycleEventToCycleForkJoinTransition;
   protected NSFForkJoinTransition cycleForkJoinToCompleteForkJoinTransiiton;
   protected NSFExternalTransition completeForkJoinToWaitForCycleEventTransition;

   // Subystem A Region
   // Regions and states, from outer to inner
   protected NSFInitialState subsystemAInitialState;
   protected NSFCompositeState initializeAState;
   protected NSFCompositeState cycleAState;
   // Transitions, ordered internal, local, external
   protected NSFExternalTransition subsystemAInitialToInitializeATransition;
   protected NSFExternalTransition initializeAToInitializeForkJoinTransition;
   protected NSFForkJoinTransition initializeForkJoinToCycleForkJoinARegionTransition;
   protected NSFExternalTransition cycleForkJoinToCycleATransition;
   protected NSFExternalTransition cycleAToCompleteForkJoinTransition;

   // Subystem B Region
   // Regions and states, from outer to inner
   protected NSFInitialState subsystemBInitialState;
   protected NSFCompositeState initializeBState;
   protected NSFCompositeState cycleBState;
   // Transitions, ordered internal, local, external
   protected NSFExternalTransition subsystemBInitialToInitializeBTransition;
   protected NSFExternalTransition initializeBToInitializeForkJoinTransition;
   protected NSFForkJoinTransition initializeForkJoinToCycleForkJoinBRegionTransition;
   protected NSFExternalTransition cycleForkJoinToCycleBTransition;
   protected NSFExternalTransition cycleBToCompleteForkJoinTransition;
   …
};
```

Tips:

- All incoming transitions to a fork-join must originate from different regions, except in the case of fork-join to fork-join transitions (see below).
- All outgoing transitions from a fork-join must terminate in different regions, except in the case of fork-join to fork-join transitions (see below).
- As an extension to UML 2.x, NSF provides a fork-join to fork-join transition, implemented in the class NSFForkJoinTransition.
- Each transition into the fork-join occurs as soon as it is satisfied.
- Note that the NSF Fork-Join is an amalgamation of the UML Fork and Join constructs.

3.7 Extending State Machines

In contrast to many state machine tools, extending an NSF state machine through inheritance or composition is a straightforward process. NSF does not rely on a code generation tool to extend state machines; rather, the framework itself is designed to make it easy to add states, entry/exit actions, transitions, guards, or other features.

Tips:

- Make structural changes to a state machine only when it is not running, preferably at construction time.
- Use a single thread to construct and make structural changes to a state machine.

3.7.1 Inheritance

Extending an NSF state machine through inheritance is as simple as deriving a new state machine class from the base state machine, and then defining the new features therein. As an example, suppose we want to extend the previously described CommandProcessor state machine such that it will automatically retry sending the command after the first response timeout. This extension is illustrated below:

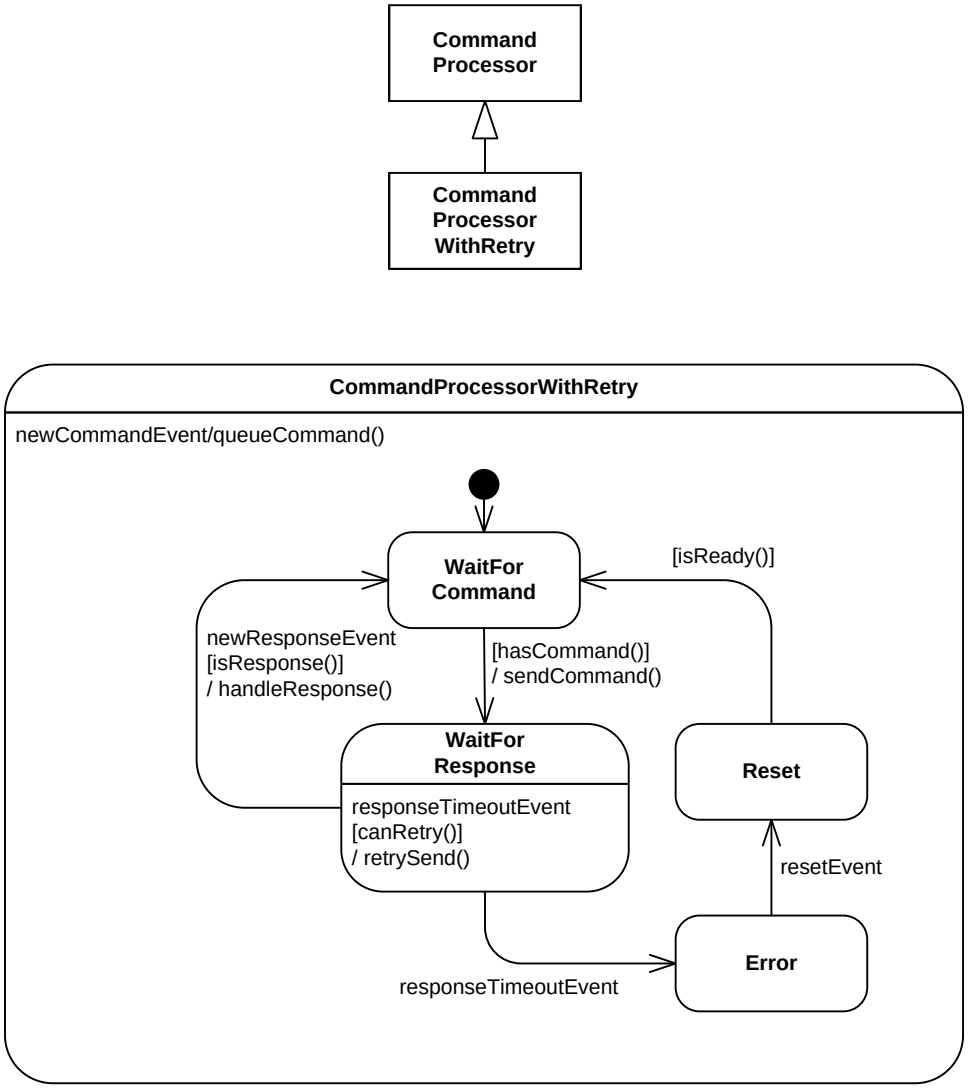

Figure 3-6: Command Processor With Retry Example

To accomplish this change, we simply derive the CommandProcessorWithRetry class from the CommandProcessor class, and include the internal transition and guard. The source code to implement the new class is shown below.

```
public class CommandProcessorWithRetry : CommandProcessor
{
   public CommandProcessorWithRetry(String name)
      : base(name)
   {
      waitForResponseReactionToResponseTimeout =
        new NSFInternalTransition("WaitForResponseReactionToResponseTimeout",
        waitForResponseState, responseTimeoutEvent, canRetry, retrySend);
      waitForResponseState.EntryActions += waitForResponseEntryActions;
   }

   private int retries = 0;
   private int maxRetries = 1;

   private NSFInternalTransition waitForResponseReactionToResponseTimeout;

   private void retrySend(NSFStateMachineContext context)
   {
      Console.WriteLine("Retry send …");
      ++retries;
      // Code to re-send the command goes here
      // …

      // Schedule timeout event, in case no response received
      responseTimeoutEvent.schedule(responseTimeout, 0);
   }

   private void waitForResponseEntryActions(NSFStateMachineContext context)
   {
      retries = 0;
   }

   private bool canRetry(NSFStateMachineContext context)
   {
      return (retries < maxRetries);
   }
}
```

3.7.2 Composition

Another way to extend NSF state machines is via composition, using a powerful technique which facilitates which we call "pluggable state machine strategies". There are two mechanisms for implementing state machine composition. The first mechanism is to use an externally defined state machine in place of a state, and the second is to plug in an externally defined state machine as a "submachine" within an existing NSFCompositeState.

3.7.2.1 State Machines as States

The first type of composition is possible because the NSFStateMachine class inherits from the NSFState class. This allows state machines to be treated just like any other state from a construction perspective. As an example of this type of composition, suppose there is an existing state machine that we wanted to include as part of the CommandProcessor to perform reset behavior after exiting the error state. The state machine might look as follows.

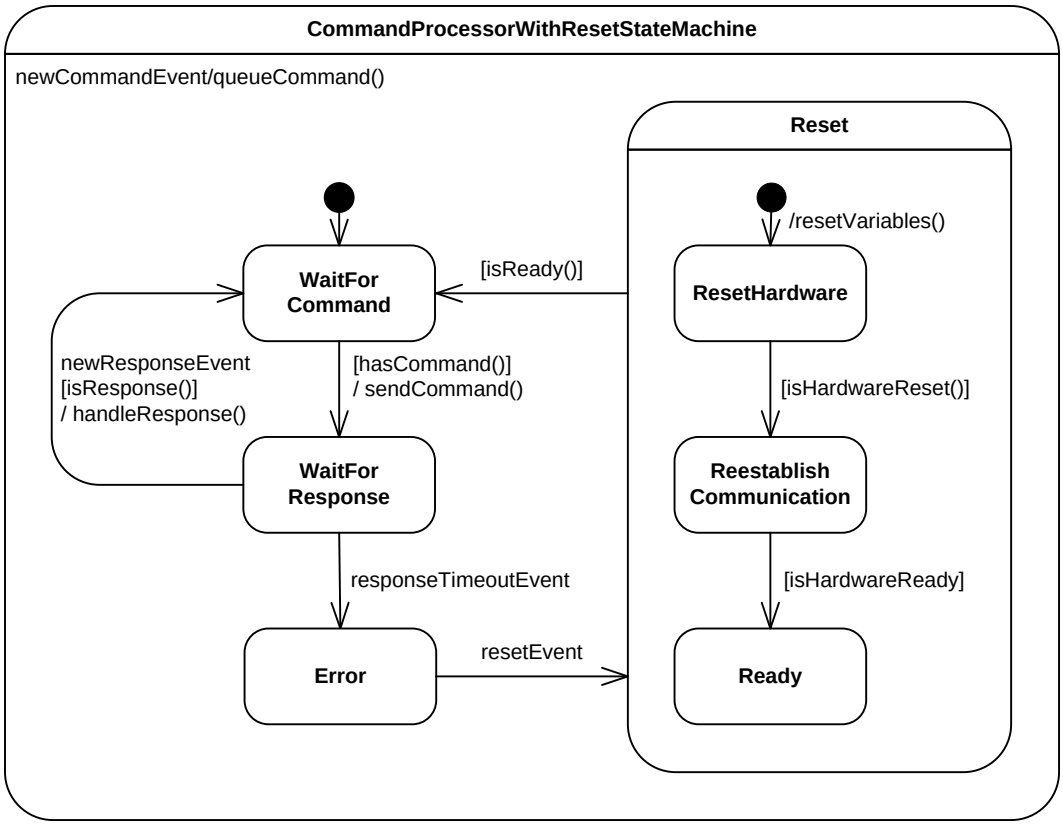

Figure 3-7: Command Processor With Reset State Machine Example

The source code to declare this state machine is as follows. Note how the ResetStrategy state machine is treated just like any other state during construction.

```
public class CommandProcessorWithResetStateMachine : NSFStateMachine
{
   …
   public CommandProcessorWithResetStateMachine(String name);
   …
   // Events
   protected NSFDataEvent<string> newCommandEvent;
   protected NSFDataEvent<string> newResponseEvent;
   protected NSFEvent responseTimeoutEvent;
   protected NSFEvent resetEvent;

   // Regions and states, from outer to inner
   protected NSFInitialState initialCommandProcessorState;
   protected NSFCompositeState waitForCommandState;
   protected NSFCompositeState waitForResponseState;
   protected NSFCompositeState errorState;
   // The ResetStrategy state machine is used as a state within this state machine.
   protected ResetStrategy resetState;

   // Transitions, ordered internal, local, external
   protected NSFInternalTransition reactionToNewCommand;
   protected NSFExternalTransition initialCommandProcessorToWaitForCommandTransition;
   protected NSFExternalTransition waitForCommandToWaitForResponseTransition;
   protected NSFExternalTransition waitForResponseToWaitForCommandTransition;
   protected NSFExternalTransition waitForResponseToErrorTransition;
   protected NSFExternalTransition errorToResetTransition;
   protected NSFExternalTransition resetToWaitForCommandTransition;
   …
};
```

3.7.2.2 State Machines as Pluggable Strategies

The second mechanism for implementing state machine composition is to plug in an externally defined state machine as a “submachine” within an existing NSFCompositeState. This technique is beneficial for designs where a base state machine describes the general behavior of several types of objects, and there exists a library of common strategies that need to be plugged into the base state machine. Not all designs benefit from this type of decomposition, but for those that do, this feature that saves enormous amounts of development and debug time.

A state machine and source code illustrating this technique is shown below. This state machine is behaviorally identical to the state machine in the previous section, but illustrates a different mechanism for implementation.

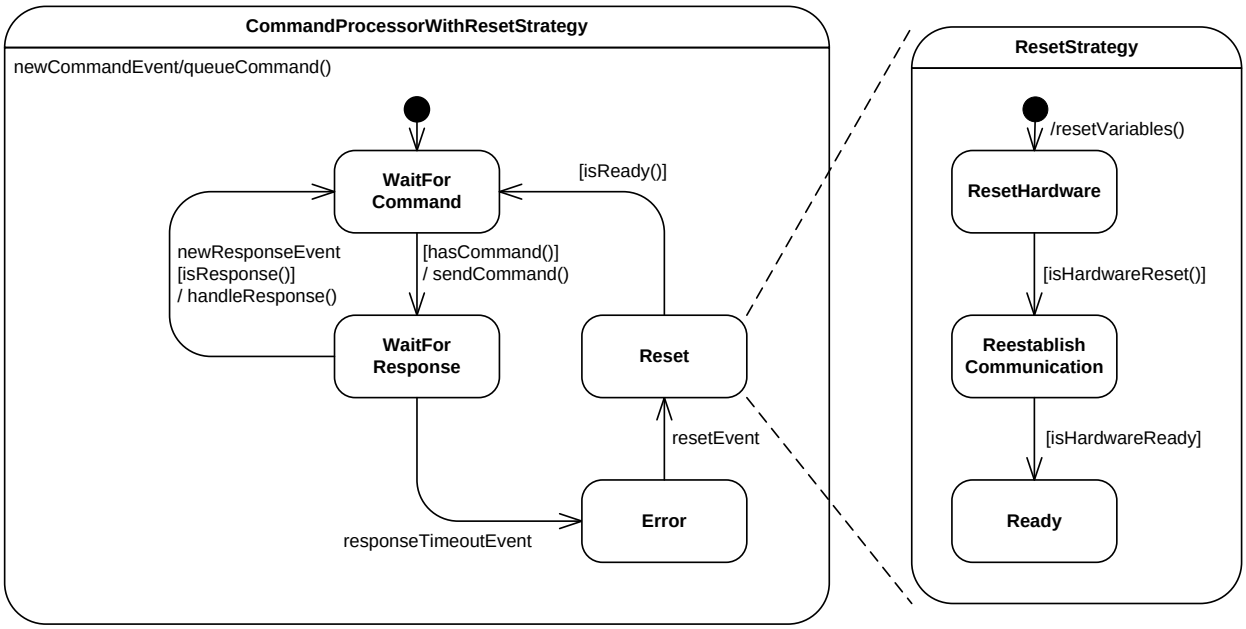

Figure 3-8: Command Processor With Reset Strategy Example

```
public class CommandProcessorWithResetStrategy : CommandProcessor
{

   public CommandProcessorWithResetStrategy(String name)
      : base(name)
   {
      resetStrategy = new ResetStrategy("ResetStrategy", resetState);
   }
   …
   // The ResetStrategy state machine is a member of the derived class.
   // It is plugged into the Reset state during initialization.
   private ResetStrategy resetState;

   private bool isReady(NSFStateMachineContext context)
   {
      return resetStrategy.ReadyState.isActive();
   }
};
```

For designs that lend themselves to this type of composition, this feature allows for a tremendous amount of reuse, thus saving time and reducing defects.

3.8 Tracing

NSF contains built-in support for tracing the run-time behavior of your application. The NSFTraceLog class implements this functionality. To turn off tracing (it is enabled by default), make the call:

```
NSFTraceLog.PrimaryTraceLog.Enabled = false;
```

The base tracing functionality is to automatically log event queueing and state entry. In addition, applications can log their own traces to record information of interest through the addTrace(…) methods. Finally, the trace file can be saved to an xml file at any time by calling the saveLog(…) method. Following is an excerpt from an example trace file.

```
<TraceLog>
  …
  <Trace>
    <Time>6531</Time>
    <EventQueued>
      <Name>NewCommandEvent</Name>
      <Source>CommandProcessor</Source>
      <Destination>CommandProcessor</Destination>
    </EventQueued>
  </Trace>
  …
  <Trace>
    <Time>6547</Time>
    <StateEntered>
      <StateMachine>CommandProcessor</StateMachine>
      <State>WaitForResponseState</State>
    </StateEntered>
  </Trace>
  <Trace>
    <Time>7031</Time>
    <EventQueued>
      <Name>NewResponseEvent</Name>
      <Source>CommandProcessor</Source>
      <Destination>CommandProcessor</Destination>
    </EventQueued>
  </Trace>
  <Trace>
    <Time>7047</Time>
    <StateEntered>
      <StateMachine>CommandProcessor</StateMachine>
      <State>WaitForCommandState</State>
    </StateEntered>
  </Trace>
  …
</TraceLog>
```

Each individual trace within the log has the following format (where text with the square brackets is trace specific).

```
<Trace>
   <Time>[time in mS since application began]</Time>
   <[trace type]>
      <[tag1]>[data1]</[tag1]>
      …
   </[trace type]>
</Trace>
```

In order to keep the trace file from growing indefinitely, the maximum number of traces in the log is capped at a user specifiable number, with the default being 5000 traces. After the maximum number of traces is reached, each new trace added to the log results in the removal of the oldest trace in the log.

Tracing is an invaluable tool for understanding the run-time behavior of your application. The xml format of the trace file makes it easy to integrate with visualization tools or spreadsheet applications such as Microsoft Office Excel™.

Tips:

- If your application permits, keep tracing turned on so that you have the trace information should it be needed.

3.9 Exception Handling

The NSF strategy for handling run-time exceptions is to capture all exceptions, so that the application stays up and running, and to provide a mechanism to notify the application of exceptions when they occur. Exceptions during construction of NSF classes are not caught, as most construction activity occurs during application startup.

Exceptions are caught by NSF classes that implement delegates, threads, state machines, and trace logs. This approach ensures that a bad action, event, guard, etc won't take down a state machine, or worse, the entire application. In all cases, the exception is caught, and then passed to a parent object, along with context information. If there is no parent object to pass the exception information on to, it is passed to the global exception handler, NSFExceptionHandler. In this manner, all run-time exceptions eventually propagate up to the global exception handler, which logs a trace, optionally saves a trace file, and calls any application exception actions registered with its ExceptionActions list.

Exception actions follow the same delegate pattern used for state entry and exit actions, with a slightly different method signature. The method signature for an exception action is:

```
void methodName(NSFExceptionContext context);
```

The following example illustrates how to register an exception action with the global exception handler.

```
static void Main(String[] args)
{
   …
   // Set up global exception handler
   NSFExceptionHandler.ExceptionActions += globalHandleException;
   …
}

static void globalHandleException(NSFExceptionContext context)
{
   Console.WriteLine("Global exception caught: " + context.Exception.ToString());
}
```

In addition to the global exception handler, there are three other locations where an application can register an exception action: NSFEventThread.ExceptionActions, NSFTimerThread.ExceptionActions, and NSFStateMachine.ExceptionActions.

3.10 Shutting Down an Application

NSF applications typically involve many threads which need to be terminated before shutting down the application. To facilitate coordinated shutdown, the framework provides a utility method called terminate() in the NSFEnvironment class. This method terminates all the state machines, event handlers, and threads in a coordinated fashion. An application should call NSFEnvironment.terminate() prior to exiting to allow the the .Net environment to garbage collect and shutdown the application in a timely fashion.

4 Porting to a New Platform

NSF in C# V2.0 is written against the .Net Framework 2.0. Currently, it is unlikely that NSF in C# would require porting the code to run on a different platform (for example, Mono). Nonetheless, to support future needs and/or platform specific features, NSF contains a platform abstraction layer that consists of three classes: NSFOSSignal, NSFOSThread, and NSFOSTimer.

The process of porting NSF to a new platform involves creating concrete classes derived from these base classes, defining constructors and bodies for their abstract properties and methods, and adding the new class constructors to the base class factory method for creating the concrete classes.

4.1 NSFOSSignal

NSFOSSignal is the base class defining objects that provide a mechanism to block a thread until a signal is sent. It contains four abstract methods that must be provided by the derived class.

```
/// <summary>
/// Represents a signal that may be used to block thread execution until
/// the signal is sent.
/// </summary>
public abstract class NSFOSSignal : NSFTimerAction
{
   …
   /// <summary>
   /// Clears the signal.
   /// </summary>
   /// <remarks>
   /// This method sets the signal to a non-signaled state,
   /// so that the next wait will block until a send is called.
   /// </remarks>
   public abstract void clear();

   /// <summary>
   /// Sends the signal.
   /// </summary>
   public abstract void send();

   /// <summary>
   /// Waits for signal to be sent.
   /// </summary>
   /// <returns>True if the signal was sent, false otherwise.</returns>
   /// <remarks>
   /// This method will block the calling thread until the signal is sent.
   /// Multiple threads must not wait on the same signal.
   /// </remarks>
   public abstract bool wait();
```

```
   /// <summary>
   /// Waits for up to <paramref name="timeout"/> milliseconds for a signal to be sent.
   /// </summary>
   /// <param name="timeout">The maximum number of milliseconds to wait.</param>
   /// <returns>True if the signal was sent, false otherwise.</returns>
   /// <remarks>
   /// This method will block the calling thread until the signal is sent or
   /// the timeout occurs.  Multiple threads must not wait on the same signal.
   /// </remarks>
   public abstract bool wait(Int32 timeout);
   …
}
```

The derived class should also provide a constructor with the following signature (illustrated from the .Net implementation).

```
/// <summary>
/// Creates a signal.
/// </summary>
/// <param name="name">The name of the signal.</param>
public NSFOSSignal_dotNet(NSFString name)
   : base(name)
{
}
```

4.2 NSFOSThread

NSFOSThread is the base class defining objects that provide a thread of execution. It contains three abstract properties and methods methods that must be provided by the derived class.

```
/// <summary>
/// Represents a general purpose thread.
/// </summary>
public abstract class NSFOSThread : NSFTaggedObject
{
   …
   /// <summary>
   /// Gets or sets the priority of the thread.
   /// </summary>
   /// <remarks>
   /// This property specifies the priority of the thread.
   /// </remarks>
   public abstract int Priority { get; set; }
```

```
   /// <summary>
   /// Gets the operating system specific thread id.
   /// </summary>
   /// <returns>The OS specific thread id.</returns>
   public abstract int OSThreadId { get; }
   …
   /// <summary>
   /// Starts the thread by calling its execution action.
   /// </summary>
   public abstract void startThread();
   …
}
```

The derived class should also provide two constructors with the following signatures (illustrated from the .Net implementation).

```
/// <summary>
/// Creates a thread.
/// </summary>
/// <param name="name">The name for the thread.</param>
/// <param name="executionAction">The action executed by the thread.</param>
/// <returns>The new thread.</returns>
/// <remarks>
/// The thread execution action is typically an execution loop.
/// When the action returns, the thread is terminated.
/// </remarks>
public NSFOSThread_dotNet(NSFString name, NSFVoidAction<NSFContext> executionAction)
    : this(name, executionAction, (int)ThreadPriority.Normal)
{
}

/// <summary>
/// Creates a thread.
/// </summary>
/// <param name="name">The name for the thread.</param>
/// <param name="executionAction">The action executed by the thread.</param>
/// <param name="priority">The priority of the thread.</param>
/// <returns>The new thread.</returns>
/// <remarks>
/// The thread execution action is typically an execution loop.
/// When the action returns, the thread is terminated.
/// </remarks>
public NSFOSThread_dotNet(NSFString name, NSFVoidAction<NSFContext> executionAction,
  int priority)
    : base(name, executionAction)
{
   …
}
```

The derived class must call the base class method executeAction() from its thread entry method. This call will pass control to the executionAction specified in the constructor.

Finally, several static properties and methods in the NSFOSThread base class might need to be completed for the new platform. These methods are:

```
/// <summary>
/// Represents an operating system thread.
/// </summary>
public abstract class NSFOSThread : NSFTaggedObject
{
   /// <summary>
   /// Gets the value for the highest priority thread.
   /// </summary>
   /// <returns>The highest thread priority.</returns>
   /// <remarks>
   /// This methods returns the highest priority that an NSFOSThread may have.
   /// The system may support higher thread priorities which are reserved for system use.
   /// Only the NSFOSTimer should use this priority level.
   /// </remarks>
   public static int HighestPriority
   {
#if NSFTarget_dotNet
   get { return (int)ThreadPriority.Highest; }
#endif
   }

   /// <summary>
   /// Gets the value for a high priority thread.
   /// </summary>
   /// <returns>The high thread priority.</returns>
   /// <remarks>
   /// Use the Priority property to set a threads priority.
   /// </remarks>
   public static int HighPriority
   {
#if NSFTarget_dotNet
   get { return (int)ThreadPriority.AboveNormal; }
#endif
   }

   /// <summary>
   /// Gets the value for the lowest priority thread.
   /// </summary>
   /// <returns>The lowest thread priority.</returns>
   /// <remarks>
   /// This methods returns the lowest priority that an NSFOSThread may have.
   /// The system may support lower thread priorities which are reserved for system use.
   /// </remarks>
   public static int LowestPriority
   {
#if NSFTarget_dotNet
   get { return (int)ThreadPriority.Lowest; }
#endif
   }
```

```
   /// <summary>
   /// Gets the value for a low priority thread.
   /// </summary>
   /// <returns>The low thread priority.</returns>
   /// <remarks>
   /// Use the Priority property to set a threads priority.
   /// </remarks>
   public static int LowPriority
   {
#if NSFTarget_dotNet
   get { return (int)ThreadPriority.BelowNormal; }
#endif
   }

   /// <summary>
   /// Gets the value for a medium priority thread.
   /// </summary>
   /// <returns>The medium thread priority.</returns>
   /// <remarks>
   /// Use the Priority property to set a threads priority.
   /// </remarks>
   public static int MediumPriority
   {
#if NSFTarget_dotNet
   get { return (int)ThreadPriority.Normal; }
#endif
   }
   …
   /// <summary>
   /// Sleeps the calling thread for the specified number of milliseconds.
   /// </summary>
   /// <param name="sleepTime">Time to sleep in milliseconds.</param>
   /// <remarks>
   /// This method sleeps the calling thread, not the thread object on which it is called.
   /// A sleep time of zero has OS defined behavior.
   /// </remarks>
   public static void sleep(UInt32 sleepTime)
   {
#if NSFTarget_dotNet
   Thread.Sleep((int)sleepTime);
#endif
   }
   …
}
```

4.3 NSFOSTimer

NSFOSTimer is the base class defining objects that provide a timing mechanism. It contains one abstract method and one abstract property that must be provided by the derived class.

```
/// <summary>
/// Represents a simple timer.
/// </summary>
public abstract class NSFOSTimer : NSFTaggedObject
{
   …
   /// <summary>
   /// Gets the current time in milliseconds since the timer was created.
   /// </summary>
   /// <returns>The current time in milliseconds.</returns>
   public abstract NSFTime CurrentTime { get; }
   …
   /// <summary>
   /// Waits for the next tick period.
   /// </summary>
   public abstract void waitForNextTick();
   …
}
```

The derived class should also provide a constructor with the following signature (illustrated from the .Net implementation).

```
/// <summary>
/// Creates a timer.
/// </summary>
/// <param name="name">The name of the timer.</param>
/// <param name="tickPeriod">The tick period of the timer.</param>
/// <remarks>
/// The tick period defines the resolution of the timer,
/// and is used by the method waitForNextTick().
/// </remarks>
public NSFOSTimer_dotNet(NSFString name, UInt32 tickPeriod)
    : base(name, tickPeriod)
{
   …
}
```

4.4 Factory Methods

Each of the platform abstraction layer classes provide a factory method, called create(), to create objects of the derived class types. The final step in porting is to add calls to the derived class constructors in the create methods of each base class, as illustrated below.

```
/// <summary>
/// Creates a signal.
/// </summary>
/// <param name="name">The name of the signal.</param>
public static NSFOSSignal create(NSFString name)
{
#if NSFTarget_dotNet
    return new NSFOSSignal_dotNet(name);
#elif NSFTarget_AnotherPlatform
    return new NSFOSSignal_AnotherPlatform(name);
#endif
}
```

After completing these steps, the framework is ready to be compiled for the target platform.

5 Conclusion

We thank you for your interest in North State Software and the North State Framework. As you work with the framework, please share your experiences with us at: http://www.northstatesoftware.com.