SDD

Wayne.Lib.StateEngine

Table of Contents

1 Document information 4
1.1 Revision history 4
1.2 Purpose and scope 4
1.3 References 4
2 Introduction 5
3 StateEngine class library 7
3.1 State class 7
3.2 PseudoState class 7
3.3 InitialState class 7
3.4 CompositeState class 7
3.5 FinalState class 7
3.6 StateMachine class 7
3.6.1 Threaded State machine 8
3.6.2 Synchronized state machine 8
3.7 Event handling 8
4 Tutorial 9
4.1.1 Modelling the Microwave oven 9
4.1.2 Writing the code 9
4.1.3 Exchanging HeatingState with a compositestate. 14
5 Features 16
5.1 Timer 16
5.2 Built-in transitions 16
5.2.1 BasicTransitionType.Init 16
5.2.2 BasicTransitionType.Done 16
5.2.3 BasicTransitionType.Error 16
5.3 AnyState as source state in transition setup. 16
5.4 ExplicitTransition 16
5.5 Generic event 17
5.6 Generic states 17
5.7 Description attributes 18
6 Diagrams 19
6.1 State Machine 19
6.2 State Engine Events 19
6.3 State Classes 20
6.4 Generic state classes 20
6.5 Transition classes 21
6.6 Timer classes 21
6.7 Description attribute classes 22
7 Namespace Wayne.Lib.StateEngine 23
7.1 Interfaces 25
7.1.1 Interface IEventConsumer 25
7.1.2 Interface IStateFactory 26
7.2 Classes 26
7.2.1 Class AnyState 26

7.2.2 Class CompositeState 26
7.2.3 Class EnterDescriptionAttribute 27
7.2.4 Class EventDescriptionAttribute 28
7.2.5 Class ExplicitTransition 29
7.2.6 Class FinalState 29
7.2.7 Class GenericEvent 30
7.2.8 Class GenericEvent`1 30
7.2.9 Class ImageDescriptionAttribute 31
7.2.10 Class InitialState 31
7.2.11 Class KeywordDescriptionAttribute 32
7.2.12 Class LogEventArgs 32
7.2.13 Class PseudoState 33
7.2.14 Class State 33
7.2.15 Class StateChangedEventArgs 35
7.2.16 Class StateDescriptionAttribute 36
7.2.17 Class StateEngineEvent 37
7.2.18 Class StateEngineException 37
7.2.19 Class StateEntry 37
7.2.20 Class StateFactories 38
7.2.21 Class StateMachine 39
7.2.22 Class StateTransitionLookup 42
7.2.23 Class TimeoutDescription 45
7.2.24 Class Timer 45
7.2.25 Class TimerEvent 46
7.2.26 Class Transition 46
7.2.27 Class TransitionInfo 47
7.3 Enumerations 48
7.3.1 Enumeration BasicTransitionType 48
7.3.2 Enumeration HistoryType 48
7.3.3 Enumeration LogType 48
7.3.4 Enumeration StateDescriptionType 48
7.3.5 Enumeration StateMachineType 49
7.3.6 Enumeration StateNameKind 49
7.3.7 Enumeration StateType 49
7.4 Delegates 49
8 Namespace Wayne.Lib.StateEngine.Description 50
8.1 Classes 50
8.1.1 Class AsyncDoneDescription 50
9 Namespace Wayne.Lib.StateEngine.Generic 51
9.1 Interfaces 51
9.1.1 Interface IGenericState`1 51
9.2 Classes 51
9.2.1 Class AsyncWorkState`1 51
9.2.2 Class CompositeState`1 53
9.2.3 Class FinalState`1 53
9.2.4 Class InitialState`1 53
9.2.5 Class PseudoState`1 53
9.2.6 Class State`1 53
9.2.7 Class TimeoutState`1 54
9.3 Enumerations 55
9.3.1 Enumeration GenericEventType 55

1 Document information

File: SDD_Wayne.Lib.StateEngine.doc

1.1 Revision history

Revision	*Date/Sign*	*Change description*
1.0	Roger Månsson	Created
1.0b	Roger Månsson	Corrected the examples.
1.0c	2006-01-23 Mattias Larsson	Corrected the examples.
1.1	2006-02-03 Roger Månsson	Renamed Event class -> StateEngineEvent. Renamed BasicTransitionTypes -> BasicTransitionType
1.2	2006-07-18	Major changes to the Transition and event handling.
1.3	2006-11-30 Mattias Larsson	Document format update.
1.4	2008-02-19	Updated document with latest changes to library.
1.5	2008-03-12 Roger Månsson	Updated examples to reflect current code and best practices.

1.2 Purpose and scope

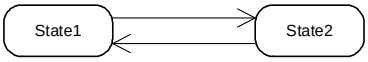
The purpose of this document is to briefly describe the design of the state engine class library and to show how the different features of it can be used. The intended audience is members of the development team.

1.3 References

2 Introduction

The state engine design is based on the theories of FSM (Finite State Machine). It is a model of an object that has an ability to be in different *state*s, and depending on its state it reacts differently to events from surrounding systems. The change between states is called *Transitions*.

There are different interpretations of the FSM theory, and the naming varies. In this introduction, the theory basis for the state engine class library will be described.



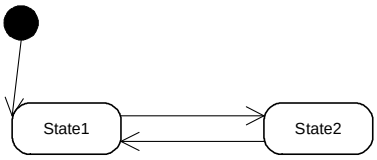
There are some state kinds.

State
This is an ordinary state where the object can be. It can make transitions to other states with a transition. A state has three possibilities to react to events from the surrounding systems:
Enter - executes when the state is entered
Exit - executes when the state is left
HandleEvent- executes when something is sent from a surrounding system. The state machine determines, depending on the current state, which action to take based on the event.
UML Notiation: A rounded rectangle

PseudoState
A pseudo state is a state that not really is a state, because the object cannot be in this state, it must directly make a transition to another state.

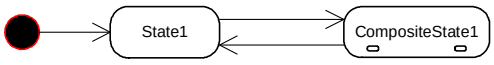
UML notation: Can be a rhomb ◇ if it is a choice state.

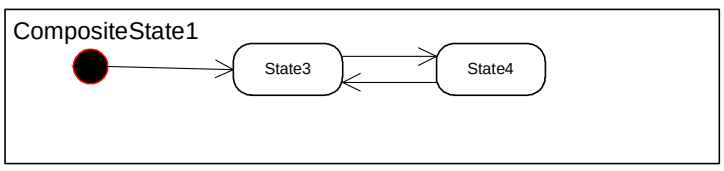
InitialState
Initial state is a start point of a state machine. The initial state is a special case of the Pseudo state, and therefore state machine cannot stay in this state. It must transition directly to another state. A state machine must contain an Initial state; otherwise it won't know where to begin.



UML Notation: A filled circle.

CompositeState
A composite state is a state that has substates. When the object is in this state, its behaviour is determined by not only what state it is in, but also what state the state is in. In fact, there is a complete new state machine contained in the composite state, and a separate state chart can be drawn for the inner state machine.





When entering a composite state we can specify three different history types.
History type none - default. When entering the composite state, it always starts at its initial state.

History type shallow - The composite state returns to the sub state it was in when it was last left. UML Notation: A circle with an H in: 
History type deep - The composite state will, like in the shallow history type, return to the state it was in when it last exited, but if that state is another composite state, it will enter its substate too as far as it goes. UML notaion: A circle with an H and an asterisk 

FinalState
The final state marks the endpoint of a state machine. It is mostly used within composite states to mark the end of a processing. The state machine cannot transition from the final state.

UML Notiation: A circle containing another, filled, circle. 

3 StateEngine class library

The StateEngine library is an implementation of the theories described in the introduction. The basis for the design is the following

- Each state is represented by an object.
- Input to the state machine is event objects of the class StateEngineEvent.
- An event is interpreted by the current state, which sends a Transition object to the State machine.
- The state machine keeps a lookup table that finds the next state based on the transition type.

3.1 State class

When implementing code for a state, the StateEngine.State class is inherited. It has three methods that can be overridden:

- Enter
- Exit
- HandleEvent

It is optional to call the base version of these methods.
The state class is the base class for all other state classes.

3.2 PseudoState class

The PseudoState class is used when the state machine cannot stay in this state, but must transition directly. This is used when we want to interpret something in the environment to determine which transition to proceed with. We cannot use Enter, Exit or HandleEvent in PseudoState. Instead the method `CreatePseudoStateTransition` is mandatory to override, since it is an abstract method.

3.3 InitialState class

The Initial state is a pseudo state that actually not has any new functionality from the base class. There should be one and only one initial state in a state machine, and it is automatically entered when starting up the machine.

3.4 CompositeState class

Composite state is a special state that owns a separate state machine that represents a substate of the state. It is possible to override Enter, Exit, but not HandleEvent. It is used internally. Instead, override the methods `BeforeHandleEvent and UnhandledEvent.`
The statemachine that is contained must be set up in the same way as any state machine (see statemachine description)

3.5 FinalState class

Final state represent an endpoint in a state machine. There can not be any transitions from this state to any other state in the same state machine. There can be several final states in one state machine. When the Final state has been reached, and after the Entry, the state engine
will automatically post a BasicTransitionType.Done transition to the state machine above the state machine containing the final state. By overriding the Enter method of a final state, the default transition can be overridden with a custom transition.

3.6 StateMachine class

The state machine class is the engine of the state machine, and owns the states. It must also be associated with one or more state factories that can produce the state objects. Normally it should not be necessary to derive from the StateMachine class, it should be enough to instantiate and configure it.

A statemachine object can only be created through the static factory method StateMachine.Create(). Here the user specifies the implementation that is required. There are two implementations for the StateMachine class that executes differently.

3.6.1 Threaded State machine

The threaded state machine has its own thread, where all state machine code executes. All execution in any state machine is triggered by an event. When the event is delivered to the state machine by some other thread, it is queued and processed by the state machine thread.

3.6.2 Synchronized state machine

This implementation does not contain any thread of its own. Instead it is executing in the thread that delivered the event. If one thread is already is executing when another thread delivers an event, the event is just queued. The event will then be handled by the thread that was running. So the impact for threads to deliver events to a state machine is less predictable than in the threaded state machine. The upside of using the synchronous state machine is that it consumes less system resources, since it does not have to maintain a separate thread.

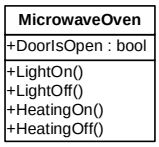
3.7 Event handling

When an event is sent into the state machine it will first be queued for handling. The state machine will then pick out the event from the queue and send it to the current state. If the current state does not mark the event as handled, the event will be put on a resend queue. When the state changes, all events in the resend queue will be moved to the normal incoming event queue. This is a prioritized queue where the event's priority and the arrival time will determine the handling order. This makes it absolutely necessary to set the Handled flag in the event objects when they are handled, otherwise they will live forever in the state machine.

4 Tutorial

4.1.1 Modelling the Microwave oven

We are going to design a Microwave oven simulation, and by that we are going to use the State Engine to implement the behavior.



It can send the following events when something changes:

DoorOpened
DoorClosed
OnButtonPressed

We want to model a very simple microwave oven without timer.

- When the door is open, or the heating is running, the light should be on.
- When the door is closed and we press the on-button we should heat.
- If the door opens during heating, the heating should be stopped.

We can identify four states:

- InitialState
- IdleState (Door is closed, and not heating)
- DoorIsOpenState (Door is open)
- HeatingState (Door is closed and the heating is on)

Now we are modeling the transitions:

State	**Event**	**Next state**
InitState	---	IdleState
IdleState	DoorOpen	DoorIsOpenState
IdleState	OnButtonPressed	HeatingState
DoorIsOpenState	DoorClosed	IdleState
HeatingState	DoorOpen	DoorIsOpenState

We are now going to translate this into a state-transition table, where the transitions are interpretations of the events in each state. Transitions are also used when we are interpreting the startup of the state machine. When we start the microwave oven, we don't know if the door is open or not, so we have to check that, and from Initstate determine if we should go to IdleState or to DoorIsOpenState.

State	**Event**	**Interpreted into Transition by state**	**Next state**
Initstate	---	DoorIsOpenTransition	DoorIsOpenState
	---	DoorIsClosedTransition	IdleState
IdleState	DoorOpen	DoorIsOpenTransition	DoorIsOpenState
IdleState	OnButtonPressed	StartCookingTransition	HeatingState
DoorIsOpenState	DoorClosed	DoorIsClosedTransition	IdleState
HeatingState	DoorOpen	StopCookingTransition	DoorIsOpenState

The Event is not needed for the state transition table, only the interpretations of it.

4.1.2 Writing the code

We create an enumeration to identify the Events

```
public enum MicroEvent
{
    DoorOpen,
    DoorClosed,
    ButtonPressed,
    Timer,
}
```

And a transition definition, also an enumeration

```
public enum MicroTransition
{
    Init,
    DoorIsOpenTransition,
    DoorIsClosedTransition,
    StartCookingTransition,
    StopCookingTransition,
    ToggleHeater
}
```

Any object such as a number or a string can be used as identifier for an event or transition type, but enumerations are strongly recommended.

We have an interface to the object that the state should be working with

```
public interface IMicrowaveOven
{
    event EventHandler OnDoorOpen;
    event EventHandler OnDoorClosed;
    event EventHandler OnButtonPressed;

    bool DoorIsOpen { get;}
    void TurnOnLight();
    void TurnOffLight();
    void TurnOnHeater();
    void TurnOffHeater();
}
```

Next, we create the state objects.

First, InitState, which should just determine if the door is open or not, and make different transitions depending on that.

```
public class InitState : InitialState
{
    // The Microwave oven to manipulate
    IMicrowaveOven oven;

    //Constructor receiving oven object from factory
    public InitState(IMicrowaveOven oven)
    {
        this.oven = oven;
    }

    //Must transition directly.
    protected override Transition CreatePseudoStateTransition(StateEntry entry)
    {
        if (oven.DoorIsOpen)
            return new Transition(this, MicroTransition.DoorIsOpenTransition);
        else
            return new Transition(this, MicroTransition.DoorIsClosedTransition);
    }
}
```

Next, we write the Idle State. Note that we now have a HandleEvent method that receives events. If other event than those expected in this state arrives, we just ignore them. The state does also contain a reference to the microwave oven object that we need to use. This is passed in to the constructor. The calls to RemovePendingEventsOfType method in Enter() is to prevent button press-events that has been created when the door is open. Otherwise they will be queued and sent to Handle event.

```
class IdleState : State
{
    // The Microwave oven to manipulate
    IMicrowaveOven oven;

    //Constructor receiving oven object from factory
    public IdleState(IMicrowaveOven oven)
    {
        this.oven = oven;
    }

    protected override void Enter(StateEntry stateEntry, ref Transition transition)
    {
        base.Enter(stateEntry, ref transition);
        RemovePendingEventsOfType(MicroEvent.ButtonPressed);
        if (oven.DoorIsOpen)
            transition = new Transition(this, MicroTransition.DoorIsOpenTransition);
    }

    protected override void HandleEvent(StateEngineEvent stateEngineEvent,
                                        ref Transition transition)
    {
        if (stateEngineEvent.Type is MicroEvent)
        {
            switch ((MicroEvent)stateEngineEvent.Type)
            {
                case MicroEvent.DoorOpen:
                    {
                        stateEngineEvent.Handled = true;
                        transition = new Transition(this,
                                    MicroTransition.DoorIsOpenTransition);
                        break;
                    }
                case MicroEvent.ButtonPressed:
                    {
                        stateEngineEvent.Handled = true;
                        transition = new Transition(this,
                                    MicroTransition.StartCookingTransition);
                        break;
                    }
            }
        }
    }

}
```

The DoorIsOpenState must execute code for both enter and exit of the state in order to turn the light on and off.

```
public class DoorIsOpenState : State
{
    // The Microwave oven to manipulate
    IMicrowaveOven oven;

    public DoorIsOpenState(IMicrowaveOven oven)
    {
        this.oven = oven;
    }

    //Code run when entering state. Turning on the light.
    protected override void Enter(StateEntry Entry, ref Transition transition)
    {
        oven.TurnOnLight();
    }

    //Handling only one event, door closed, and then we transition.
    protected override void HandleEvent(StateEngineEvent stateEngineEvent,
                                        ref Transition transition)
    {
        if (stateEngineEvent.Type.Equals(MicroEvent.DoorClosed))
        {
            stateEngineEvent.Handled = true;
            transition = new Transition(this, MicroTransition.DoorIsClosedTransition);
        }
    }

    //Code run when exiting state. Turning off the light.
    protected override void Exit()
    {
        oven.TurnOffLight();
    }
}
```

The heating state is much the same, but both light and heater should be turned on.

```
public class DoorIsOpenState : State
{
    // The Microwave oven to manipulate
    IMicrowaveOven oven;

    public DoorIsOpenState(IMicrowaveOven oven)
    {
        this.oven = oven;
    }

    //Code run when entering state. Turning on the light.
    protected override void Enter(StateEntry Entry, ref Transition transition)
    {
        oven.TurnOnLight();
    }

    //Handling only one event, door closed, and then we transition.
    protected override void HandleEvent(StateEngineEvent stateEngineEvent,
                                        ref Transition transition)
    {
        if (stateEngineEvent.Type.Equals(MicroEvent.DoorClosed))
        {
            stateEngineEvent.Handled = true;
            transition = new Transition(this, MicroTransition.DoorIsClosedTransition);
        }
    }

    //Code run when exiting state. Turning off the light.
    protected override void Exit()
    {
        oven.TurnOffLight();
    }
}
```

Now we need a state factory that we can associate with the state machine so the states can be created correctly.

```
public class MicroStateFactory : IStateFactory
{
    private IMicrowaveOven oven;

    public MicroStateFactory(IMicrowaveOven oven)
    {
        this.oven = oven;
    }

    public State CreateState(string stateName)
    {
        //Root states
        if (stateName == typeof(InitState).FullName)
            return new InitState(oven);
        else if (stateName == typeof(IdleState).FullName)
            return new IdleState(oven);
        else if (stateName == typeof(DoorIsOpenState).FullName)
            return new DoorIsOpenState(oven);
        else if (stateName == typeof(HeatingState).FullName)
            return new HeatingState(oven);
        else
            return null;
    }

    public string Name
    {
        get { return "MicroStateFactory"; }
    }
}
```

Now all we need is to assemble everything, and create a state-transition configuration. We create a class called Configuration that handles the setup of the state transition table here to illustrate it. It does also configure the state-transition table.

```
static class Configuration
{
 public static void Config(StateTransitionLookup lookup)
 {
   //InitState transitions
   lookup.AddTransition<InitState, IdleState>(MicroTransition.DoorIsClosedTransition);
   lookup.AddTransition<InitState, DoorIsOpenState>(MicroTransition.DoorIsOpenTransition);

   //IdleState transitions
   lookup.AddTransition<IdleState, DoorIsOpenState>(MicroTransition.DoorIsOpenTransition);
   lookup.AddTransition<IdleState, Heating.CompositeState>(MicroTransition.StartCookingTransition);

   //DoorIsOpenState transitions
   lookup.AddTransition<DoorIsOpenState, IdleState>(MicroTransition.DoorIsClosedTransition);

   //HeatingState transitions
   lookup.AddTransition<HeatingState, IdleState>(MicroTransition.StopCookingTransition);
 }
}
```

4.1.3 Exchanging HeatingState with a compositestate.

A microwave oven does usually not heat all the time, rather it oscillates between heating and resting with a given time interval. We want to extend the Microwave state machine so it can reflect such functionality.

We create a new state called Heating.Composite (Heating is an additional namespace). It handles turning on light when entering and turning it right off when exiting. The control over the heater is left to the sub states.

The composite state must add a state factory and configure its state for itself.

```
class CompositeState : Wayne.Lib.StateEngine.CompositeState
{
    // The Microwave oven to manipulate
    IMicrowaveOven oven;

    //Constructor receiving oven object from factory
    public CompositeState(IMicrowaveOven oven)
    {
        this.oven = oven;
        StateMachine.AddStateFactory(new MicroStateFactory(oven));
        Configuration.Config(StateMachine.StateTransitionLookup);
    }

    protected override void Enter(StateEntry stateEntry, ref Transition transition)
    {
        base.Enter(stateEntry, ref transition);
        oven.TurnOnLight();
    }

    public override void UnhandledEvent(StateEngineEvent stateEngineEvent, ref Transition transition)
    {
        base.UnhandledEvent(stateEngineEvent, ref transition);
        if (stateEngineEvent.Type.Equals(MicroEvent.DoorOpen))
        {
            stateEngineEvent.Handled = true;
            transition = new Transition(this, MicroTransition.StopCookingTransition);
        }
    }

    protected override void Exit()
    {
        base.Exit();
        oven.TurnOffLight();
        oven.TurnOffHeater(); // Turn off heater.
    }
}
```

We create one state for On and one for Off.

```
class OnState : State
{
    // The Microwave oven to manipulate
    IMicrowaveOven oven;

    //Constructor receiving oven object from factory
    public OnState(IMicrowaveOven oven)
    {
        this.oven = oven;
    }

    protected override void Enter(StateEntry stateEntry, ref Transition transition)
    {
        base.Enter(stateEntry, ref transition);
        oven.TurnOnHeater();
        ActivateTimer(new Timer(this, MicroEvent.Timer, 5000, null));
    }

    protected override void HandleEvent(StateEngineEvent stateEngineEvent,
                                        ref Transition transition)
    {
        base.HandleEvent(stateEngineEvent, ref transition);

        if (stateEngineEvent.Type.Equals(MicroEvent.Timer))
        {
            stateEngineEvent.Handled = true;
            transition = new Transition(this, MicroTransition.ToggleHeater);
        }
```

```
    }
}

class OffState : State
{
    // The Microwave oven to manipulate
    IMicrowaveOven oven;

    //Constructor receiving oven object from factory
    public OffState(IMicrowaveOven oven)
    {
        this.oven = oven;
    }

    protected override void Enter(StateEntry stateEntry, ref Transition transition)
    {
        base.Enter(stateEntry, ref transition);
        oven.TurnOffHeater();
        ActivateTimer(new Timer(this, MicroEvent.Timer, 2000, null));
    }

    protected override void HandleEvent(StateEngineEvent stateEngineEvent,
                                        ref Transition transition)
    {
        base.HandleEvent(stateEngineEvent, ref transition);

        if (stateEngineEvent.Type.Equals(MicroEvent.Timer))
        {
            stateEngineEvent.Handled = true;
            transition = new Transition(this, MicroTransition.ToggleHeater);
        }
    }
}
```

These are configured in the configuration for the composite state.

```
static class Configuration
{
    public static void Config(StateTransitionLookup sl)
    {
        sl.AddTransition<InitState, OnState>(MicroTransition.Init);
        sl.AddTransition<OnState, OffState>(MicroTransition.ToggleHeater);
        sl.AddTransition<OffState, OnState>(MicroTransition.ToggleHeater);
    }
}
```

We also modify the Configuration for the root state machine, so we reference the Heating.Composite state instead of HeatingState.

```
public static void Config(StateTransitionLookup lookup)
{
    //InitState transitions
    lookup.AddTransition<InitState, IdleState>(MicroTransition.DoorIsClosedTransition);
    lookup.AddTransition<InitState, DoorIsOpenState>(MicroTransition.DoorIsOpenTransition);

    //IdleState transitions
    lookup.AddTransition<IdleState, DoorIsOpenState>(MicroTransition.DoorIsOpenTransition);
    lookup.AddTransition<IdleState, Heating.CompositeState>(MicroTransition.StartCookingTransition);

    //DoorIsOpenState transitions
    lookup.AddTransition<DoorIsOpenState, IdleState>(MicroTransition.DoorIsClosedTransition);

    //HeatingState transitions
    lookup.AddTransition<Heating.CompositeState, IdleState>(MicroTransition.StopCookingTransition);
}
```

The State factory must also be extended so it can create the new states Heating.Composite, Heating.OnState and Heating.OffState.

Now the microwave oven will oscillate during heating between heating 5 seconds, and resting 2 seconds.

5 Features

This section describes features that have been implemented in the state engine library to help the developer create simple and intuitive applications.

5.1 Timer

A state can create and activate timers that sends timer events after a specified time.

```
public override void Enter(StateEngine.StateEntry Entry)
{
     this.ActivateTimer(new StateEngine.Timer(this, MicroEvents.Timer, 20000);
}
```

This row will activate a timer that will send a TimerEvent with the type MicroEvents.Timer to the HandleEvent method of the state. Per default, the timer will be disabled when the state is left, but it can also be configured that it is still active after the state is left, and the timer event will be sent to the state that is active when the timeout occurs. The timers will also per default only fire once, and then remove itself, but a periodic timer can also be configured.

5.2 Built-in transitions

The state engine will post a few built-in transitions in different situations.

5.2.1 BasicTransitionType.Init

The init transition is posted to the Initial state in the state machine. It is of no use to add handlers for the Init transitions, since it will only be used in that situation. Of course can the type be used for other transitions as well, but it is not recommended.

5.2.2 BasicTransitionType.Done

A CompositeState state machine that is transitioned into a FinalState will automatically post a Done transition in the parent state machine.

5.2.3 BasicTransitionType.Error

When an error occurs in the state machine the Error transition is posted. The application use this to go to error states. The state engine sends this transition when an exception has slipped out of the user-functions Enter or HandleEvent. If we get an exception in Exit, it is logged, but does not send any automatic transition.

5.3 AnyState as source state in transition setup.

In many state machines we want a state that can be entered from all states in a simple way. A good example of this is the Error transition described above. Therefore we can implement a wildcard transition that will go to a specified state whatever state we are in. It is entered in the state transition setup as the dummy state StateEngine.AnyState.

```
lookup.AddTransition<Wayne.Lib.StateEngine.AnyState, ErrorState>( BasicTransitionType.Error)
```

If another state-transition setup is configured for the same transition, it will override the wildcard transition.

5.4 ExplicitTransition

In some cases, we want to perform transitions to a specific state and short-circuit the state-transition configuration, and go directly to a known state. This cannot be used as a replacement to the state-transition configuration, since the configuration is also used to determine what state objects to create.

One example where to use explicit transition is when we are transitioned to an Error state with a general transition from AnyState described above. In some cases we want to transition back to where we came from.

When we enter the state we get a StateEntry object. This contains a reference to the source transition. That in turn has a reference to the source state. We save a reference to the source state in the error-state object. When we want to leave the error state, we create an ExplicitTransition to the state we came from.

Example

```
class ErrorState : StateEngine.State
{
    StateEngine.State cameFromState;

    public override void Enter(StateEngine.StateEntry Entry, ref  Wayne.Lib.StateEngine.Transition
transition)
    {
        base.Enter(Entry);
        cameFromState = Entry.SourceTransition.Sender; // save reference to sourcestate

        //Show a popup or something. When it is closed it sends event type
       // MicroEvents.ErrorConfirmation
        //
    }

    public override void HandleEvent(StateEngine.StateEngineEvent stateEngineEvent,
      ref Wayne.Lib.StateEngine.Transition transition)
    {
        base.HandleEvent(stateEngineEvent);
        if (stateEngineEvent.Type == MicroEvents.ErrorConfirmation)
        {
            transition=new StateEngine.ExplicitTransition(this,     //Sender
                                    MicroTransitions.Done,          //Transition type
                                     cameFromState.Name);           //Target state
        }
    }
}
```

The target state is looked up through the state name. The state is searched in the following order:
1. Search in the same state machine as this state
2. Search in all sub-state machines for this state machine.
3. Go to the parent state machine to this, and search there.

5.5 Generic event

Normally you need to add more data to an event than the event type. You can create a subclass of the StateEngineEvent class and add the additional data. If you get the data from an EventArgs object, typically in a .Net event, the GenericEvent class can be used to encapsulate this data without needing to create a new class.

```
void SomeEventFired(object sender, EventArgs e)
{
    stateMachine.IncomingEvent(new GenericEvent<EventArgs>(States.EventType.EventHappened, sender,
e));
}
```

This approach is a bridge between the .Net event handling pattern and the StateEngine event handling.

The code can be further simplified by using the static method GenericEvent.Create(), that detects the type of the EventArgs by inferred usage.

```
void SomeEventFired(object sender, EventArgs e)
{
    stateMachine.IncomingEvent(GenericEvent.Create(States.EventType.EventHappened, sender, e));
}
```

5.6 Generic states

In all cases the state objects in a state machine needs to reference an object that it represents the logic for. In the example with the Microwave oven above, in each of the state classes there is a reference to the `IMicrowaveOven` object.

The generic states are a set of state classes that has expanded the functionality of the base state classes with the possibility to have a Main object already in the state. The type of the Main object is specified as a generic

parameter to the class (Wayne.Lib.StateEngine.Generic.State<IMicrowaveOven>). This means that you always have a property named “Main” that is of the type IMicrowaveOven.

In addition to the normal state classes, there are two additional base state classes that is made for the convenience of the developer:

- TimeoutState<T> – A state class with a built-in timeout behavior
- AsyncWorkState<T> – A state class that spawns a thread from the thread pool. When the thread execution finishes, it posts a transition automatically.

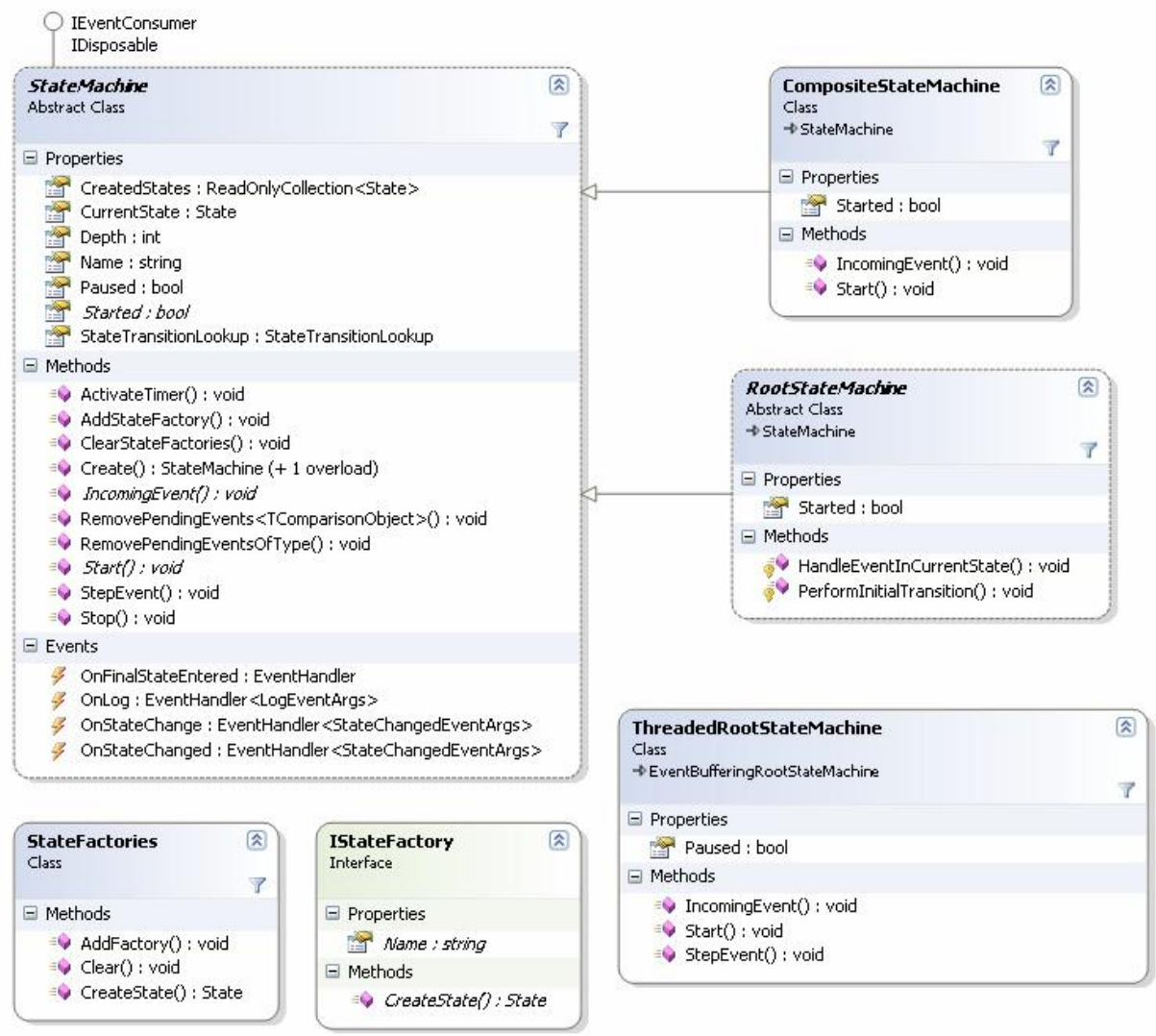
5.7 Description attributes

The description attributes is used to document a state machine and its logic. There are tools (Waynel.Lib.StateEngine.Analyzer) to analyze a state machine based on the description attributes combined with the transition tables. It is good practice to apply the description attributes to all state machines correctly. The description attributes can also be extracted to create a document that shows the state machine logic.

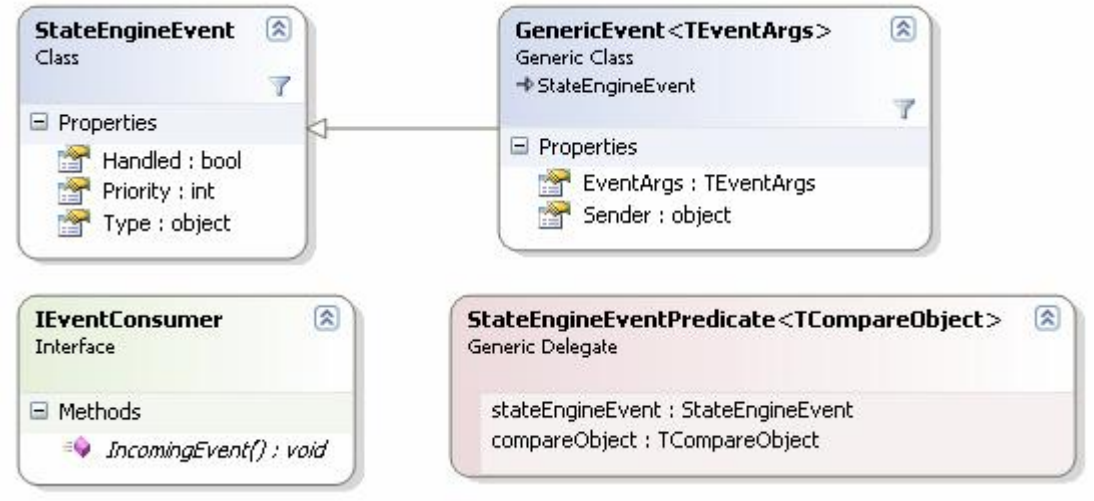
- **StateDescription** – Gives a overall description of the state and its purpose.
- **EnterDescription** – Describes what conditions that might trigger a direct transition from Enter.
- **EventDescription** – Describes wich events that are handled and if those triggers transitions.
- **KeywordDescription** – Optional description of different keywords associated with the state.
- **ImageDescription** – A way to define an image that can be automatically inserted to the statemachine documentation document.

6 Diagrams

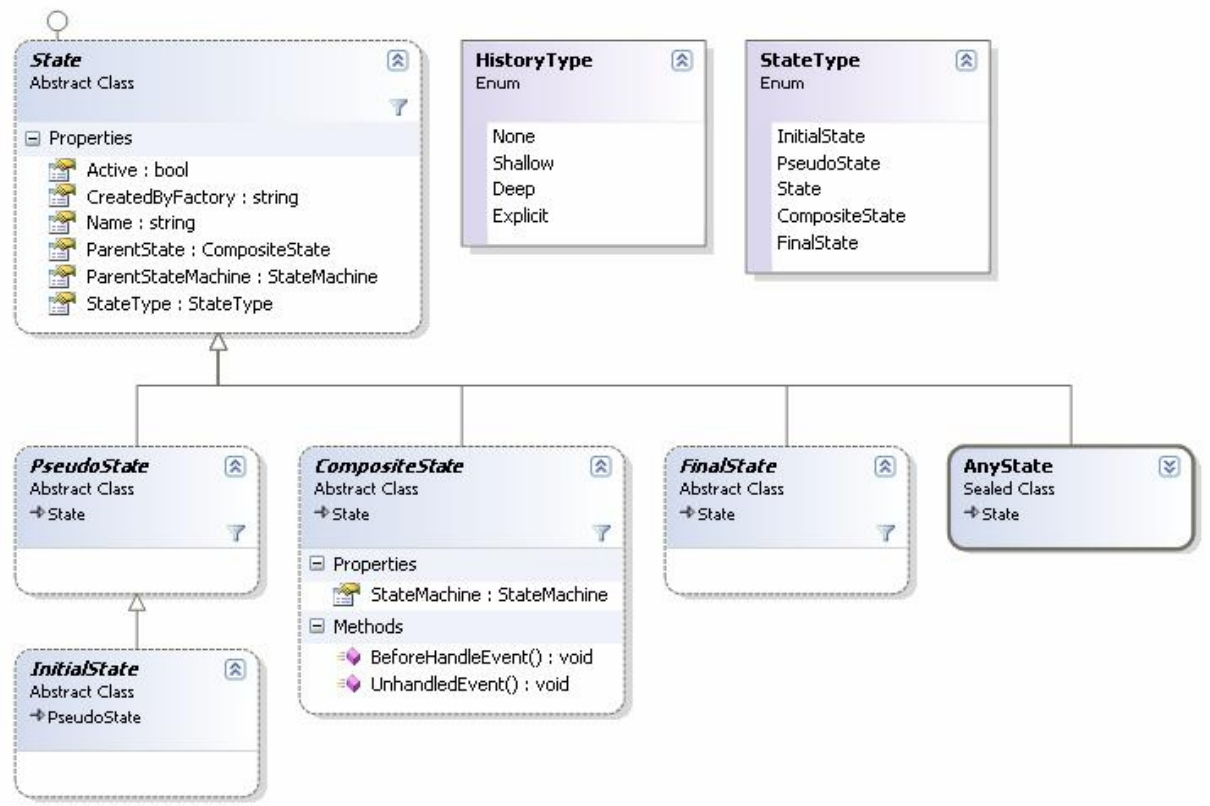
6.1 State Machine



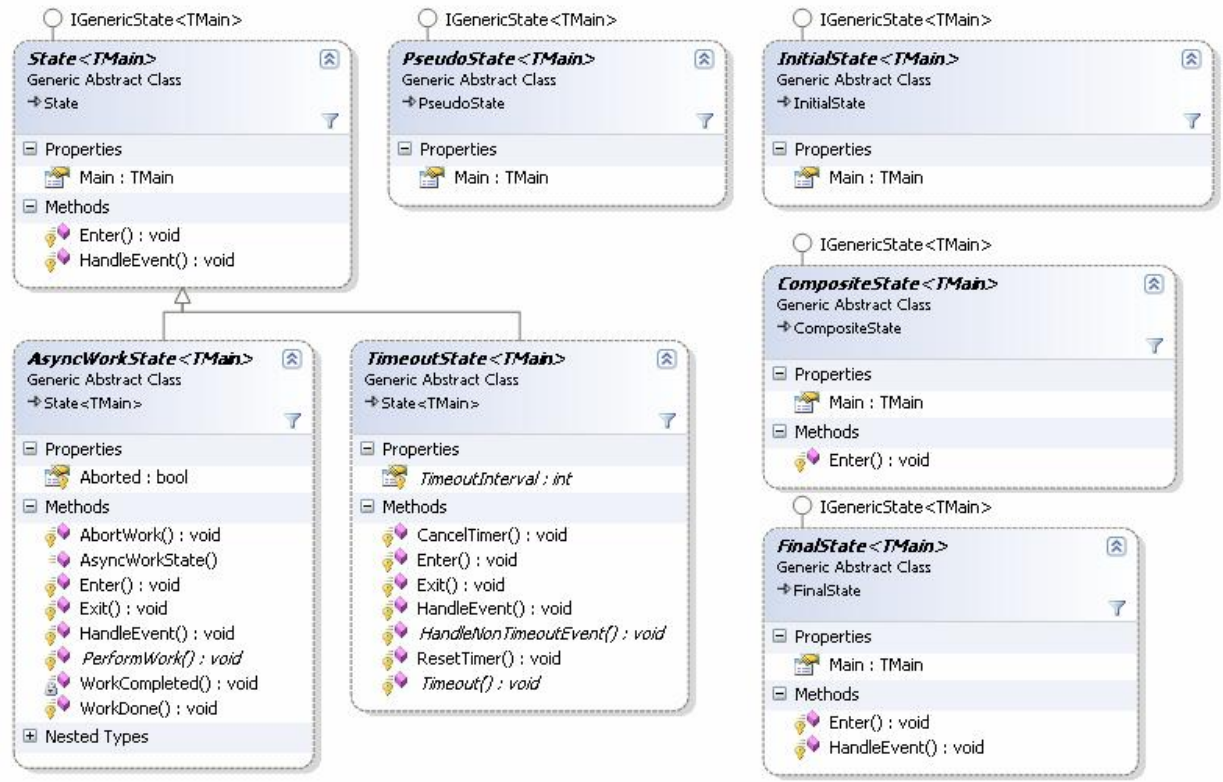
6.2 State Engine Events



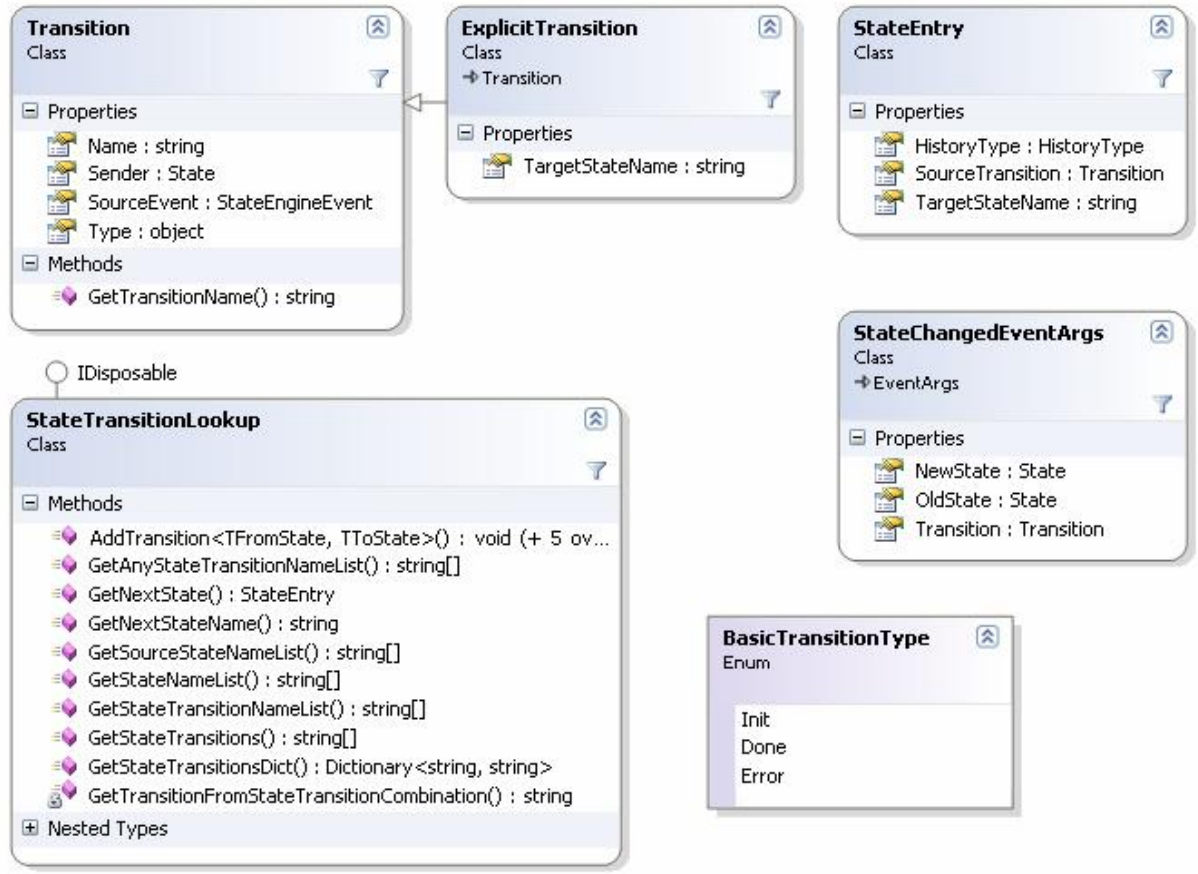
6.3 State Classes



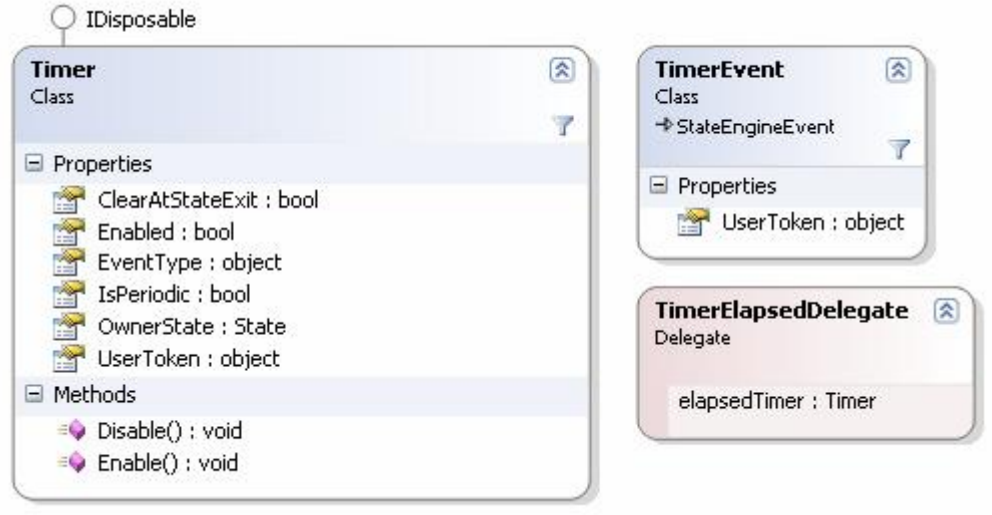
6.4 Generic state classes



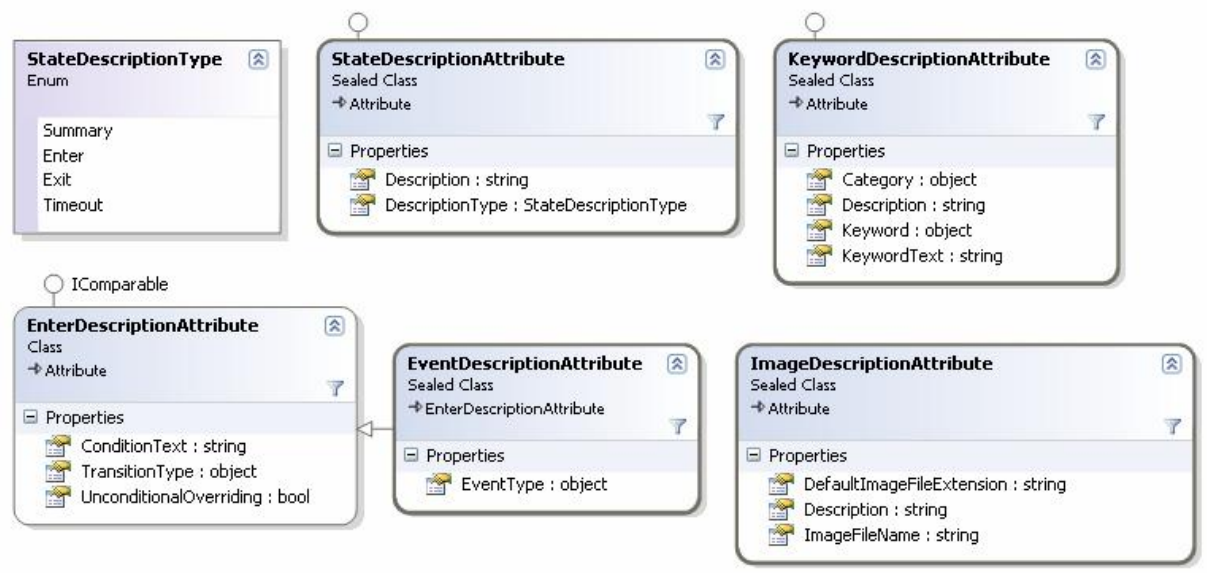
6.5 Transition classes



6.6 Timer classes



6.7 Description attribute classes



7 Namespace Wayne.Lib.StateEngine

Interfaces

IEventConsumer	Interface from the State machine to event producers that they can use to send events to the machine.
IStateFactory	Interface for State Factories. The factories are added to the state machine, and are used to create the actual state objects. The Wayne.Lib.StateEngine library does not provide any implementation for this interface, it must be implemented in each application.

Classes

AnyState	This is used to define the sourcestate as any state, a wildcard, if we want to create a general transition from a unspecified state in a machine.
CompositeState	Composite state is an abstract class that should be overriden to create a state that within itself have a state machine with sub states. When creating the composite state, a message handling thread should be sent in to the composite state constructor. It can either be a new thread if the composite state machine should run in a separate thread , or the thread of the parent state machine if the machine should run in the same thread.
EnterDescriptionAttribute	Describe the enter-transition for a state class.
EventDescriptionAttribute	Describe the event / transition relationship for a state class.
ExplicitTransition	The Explicit transition is a way to go around the State Transition Lookup, and directly decide which state to go to. It can be used in a small state machine application when an soft-coded state-transition configuration not is neccecary. Another usage is when doing transitions to general states like an error state. When we want to return from that state, we want to return to the state where it came from.
FinalState	Final state represent an endpoint in a state machine. There can not be any transitions from this state to any other state in the same state machine. There can be several final states in one state machine. When the Final state has been reached, and after the Entry, the state engine will automatically post a BasicTransitionTypes.Done transition to the state machine above the state machine containing the final state.
GenericEvent	Static class that can be used only as a substitute for the GenericEvent<> constructor. It allows C# developers to avoid typing the type argument by inferred parameter type resolution.
GenericEvent`1	GenericEvent is an event that can be used with .Net Events and delegates. Instead of defining each event as both an EventArgs structure and an StateEngineEvent structure with about the same content, this class is a StateEngineEvent that can

	contain information from a .Net event.
ImageDescriptionAttribute	Adds a document image describing this state.
InitialState	Initial state is the first state to be entered in a state machine. There should only be one initial state, and the state engine automatically enters that state at startup. It is a pseudo state so it must direclty transition to another state. The class does not introduce any new behaviour from pseudostate, but marks that it is an intial state.
KeywordDescriptionAttribute	Add a keyword to the state. Group the keywords in categories.
LogEventArgs	The Log Event args is used to carry log strings when logging internally in the state machine. Applications can hook on the StateMachine.OnLog event in order to catch loggins from the inner workings of the statemachine.
PseudoState	A pseudo state is a state that can be entered, but it must transition directly. The state machine can not stay in this state,but must continue directly. Therefore the Enter, and exit methods are sealed and can not be overriden. Insted, the abstract method CreatePseudoStateTransition method must be overriden. It receives the entry information and must return a transition direclty. The pseudo state can be used to implement choce states, and is the base class for initial states.
State	State is the base state of all states in the state machine.
StateChangedEventArgs	Argument in the StateChanged events. It contains the new and the old state in a state transition.
StateDescriptionAttribute	An attribute that can be applied to state classes, that is used to document the state.
StateEngineEvent	Event class that implements the Event interface. It is the base class to be used to send events to states in the state machine.
StateEngineException	Wayne.Lib.StateEngine exception is a general exception that can be thrown from the state engine library.
StateEntry	StateEntry contains information about the entry of a state. It is produced by the state transition lookup. The application receives it as an in parameter to the State.Enter() method. What is of intrest to the application might be the sourceTransition, and the attached source Event object.
StateFactories	A collection of State factories. A state machine can have states created by multiple state factories. When a state object should be created, the factories are queried one by one for the state name.
StateMachine	State machine is the core engine in the Wayne.Lib.StateEngine model. It contains the

	states, contains the information about the state-transition lookup table, and handles the external and internal events. Normally it should not be neccesary to override this class.
StateTransitionLookup	StateTransitionLookup is used by the State machine to find the state to change to when a transition occurs. It uses a dataset that contains the actual data.
TimeoutDescription	Describe the timeout / transition relationship for a state class of the generic TimeoutState.
Timer	Timer is a timer that should be used within the state machine.
TimerEvent	Event class for the Timer events.
Transition	The transition class is used by state objects to signal that a change has occured. The transition is interpreted by the statemachine's State-transition lookup table. it determines which the next state should be. The transition type identifies the transition and is supplied by the application. It is recommended that enums are used as transition types, but any type can be used.
TransitionInfo	A class wrapping information about a transition.

Enumerations

BasicTransitionType	Basic Transition Types is for automatic transitions.
HistoryType	History type defines the way that a composite state is entered.
LogType	Categorizes the log entries in the OnLog event from the State machine.
StateDescriptionType	Used in the DescriptionAttribute to categorize the different descriptions.
StateMachineType	Enumeration of the types of statemachines that can be created. Used in the factory method StateMachine.Create.
StateNameKind	The two kinds of names for a state.
StateType	The state types as an enumeration. This can be used in a future design tool.

Delegates

StateEngineEventPredicate`1	Delegate that is used to match StateEngine events.

7.1 Interfaces

7.1.1 Interface IEventConsumer

```
public interface IEventConsumer
```

Summary
Interface from the State machine to event producers that they can use to send events to the machine.
Methods

IncomingEvent

`public void IncomingEvent(Lib.StateEngine.StateEngineEvent stateEngineEvent);` Inject events into the state engine.	
stateEngineEvent	The event object that should be sent to the state machine

7.1.2 Interface IStateFactory

`public interface IStateFactory`

Summary
Interface for State Factories. The factories are added to the state machine, and are used to create the actual state objects. The Wayne.Lib.StateEngine library does not provide any implementation for this interface, it must be implemented in each application.

Properties

Name `string`	R	Name of the state facory.

Methods

CreateState `public Lib.StateEngine.State CreateState(string stateFactoryName);` Create a state object from the specified State name.	
stateFactoryName	Name of the state to be created.
Return value	If successful, it returns the object for the state name. If it not was found, it returns null.

7.2 Classes

7.2.1 Class AnyState

`public class AnyState : State`

Summary
This is used to define the sourcestate as any state, a wildcard, if we want to create a general transition from a unspecified state in a machine.

Constructors

`public AnyState();` Initializes a new instance of the StateEngine.AnyState class.

7.2.2 Class CompositeState

`abstract public class CompositeState : State`

Summary
Composite state is an abstract class that should be overriden to create a state that within itself have a state machine with sub states. When creating the composite state, a message handling thread should be sent in to the composite state constructor. It can either be a new thread if the composite state machine should run in a separate thread , or the thread of the parent state machine if the machine should run in the same thread.

Properties

StateMachine `Lib.StateEngine.StateMachine`	R	Returns the state machine that is used with the Composite state.

Constructors

`protected CompositeState();` Initializes a new instance of a composite state.

Methods

BeforeHandleEvent `public void BeforeHandleEvent(Lib.StateEngine.StateEngineEvent stateEngineEvent, ref Lib.StateEngine.Transition@ transition);` Method that can be overriden in descendant classes, so events can be handed before they are sent to the state machine.	
stateEngineEvent	
transition	Perform a transaction by assigning a transition object to this ref parameter.

Dispose `protected void Dispose(bool disposing);` Internal dispose method.	
disposing	

Finalize `protected void Finalize();` Destructor

HandleEvent `protected void HandleEvent(Lib.StateEngine.StateEngineEvent stateEngineEvent, ref Lib.StateEngine.Transition@ transition);` Handles an incoming event.	
stateEngineEvent	Event that is received.
transition	Perform a transaction by assigning a transition object to this ref parameter.

UnhandledEvent `public void UnhandledEvent(Lib.StateEngine.StateEngineEvent stateEngineEvent, ref Lib.StateEngine.Transition@ transition);` Method that can be overriden if wanting to handle events that not has been handled in the composite state machine.	
stateEngineEvent	
transition	Perform a transaction by assigning a transition object to this ref parameter.

7.2.3 Class EnterDescriptionAttribute

`public class EnterDescriptionAttribute : Attribute`

Summary
Describe the enter-transition for a state class.

Properties

ConditionText `string`	R	A descriptive text for the condition.
TransitionType `object`	R	Transition type that is performed.
UnconditionalOverriding	R	Tells whether this transition unconditionally overrides

`bool`		all inherited ones.

Constructors

`public EnterDescriptionAttribute(string conditionText, object transitionType);` Drescribe the event / transition relationship for a state class.	
conditionText	A descriptive text for the condition.
transitionType	Transition that is performed.

`public EnterDescriptionAttribute(string conditionText, object transitionType, bool unconditionalOverriding);` Drescribe the event / transition relationship for a state class.	
conditionText	A descriptive text for the condition.
transitionType	Transition that is performed.
unconditionalOverriding	Tells whether this transition unconditionally overrides all inherited ones.

Methods

CompareTo
`public int CompareTo(object obj);`
Compares this attribute with another one.

obj	

Equals
`public bool Equals(object obj);`
Equals.

obj	

GetHashCode
`public int GetHashCode();`
GetHashCode. The compiler complained that this method also should be overriden when the Equals was.

7.2.4 Class EventDescriptionAttribute

`public class EventDescriptionAttribute : EnterDescriptionAttribute`

Summary
Describe the event / transition relationship for a state class.

Properties

EventType `object`	R	Event type object.

Constructors

`public EventDescriptionAttribute(object eventType, string conditionText, object transitionType);` Describe the event / transition relationship for a state class.	
eventType	Event type object
conditionText	A descriptive text for the condition.
transitionType	Transition that is performed.

Methods

CompareTo `public int CompareTo(object obj);` Compares this attribute with another one.	
obj	

7.2.5 Class ExplicitTransition

```
public class ExplicitTransition : Transition
```

Summary

The Explicit transition is a way to go around the State Transition Lookup, and directly decide which state to go to. It can be used in a small state machine application when an soft-coded state-transition configuration not is neccecary. Another usage is when doing transitions to general states like an error state. When we want to return from that state, we want to return to the state where it came from.

Example

```
    Example of how to use the Explicit state as a "back" transition.

    class ErrorState : Wayne.Lib.StateEngine.State
    {
        string cameFromState;

        public override void Enter(Wayne.Lib.StateEngine.StateEntry Entry)
        {
            cameFromState = Entry.SourceTransition.Sender.Name;

            //Do some error handling code
        }

        public override void HandleEvent(Wayne.Lib.StateEngine.Event
EventToHandle)
        {
            if (EventToHandle is ErrorConfirmedEvent) //The appropriate event has
arrived, so we can leave the state.
            {
                PostTransition(new Wayne.Lib.StateEngine.ExplicitTransition(this,
Wayne.Lib.StateEngine.BasicTransitionTypes.Done, cameFromState));
            }
        }
    }
```

Properties

TargetStateFactoryName `string`	R	Name of the target state of the transition.

Constructors

`public ExplicitTransition(Lib.StateEngine.State sender, object transitionType, string targetStateFactoryName);` Constructor for the explicit transaction	
sender	State object that issued the transition
transitionType	Object representing the type of the transition
targetStateFactoryName	FactoryName of the state that should be searched for and entered.

7.2.6 Class FinalState

```
abstract public class FinalState : State
```

Summary
Final state represent an endpoint in a state machine. There can not be any transitions from this state to any other state in the same state machine. There can be several final states in one state machine. When the Final state has been reached, and after the Entry, the state engine will automatically post a BasicTransitionTypes.Done transition to the state machine above the state machine containing the final state.

Constructors

`protected FinalState();` Initializes a new instance of the StateEngine.FinalState class.

Methods

Enter `protected void Enter(Lib.StateEngine.StateEntry stateEntry, ref Lib.StateEngine.Transition@ transition);` Override to add Enter code to the Final State.	
stateEntry	Information about the state entry.
transition	Set to a transition object to perform a transition.

7.2.7 Class GenericEvent

```
abstract public class GenericEvent : Object
```

Summary
Static class that can be used only as a substitute for the GenericEvent<> constructor. It allows C# developers to avoid typing the type argument by inferred parameter type resolution.

Methods

Create `public Lib.StateEngine.GenericEvent{``0} Create(object eventType, object sender, eventArgs);` Static method that can be used instead of the constructor. The advantage is that by using a method C# can figure out the type parameter by the input argument, and thus decreasing the amount of code to write.	
eventType	
sender	
eventArgs	

7.2.8 Class GenericEvent`1

```
public class GenericEvent`1 : StateEngineEvent
```

Summary
GenericEvent is an event that can be used with .Net Events and delegates. Instead of defining each event as both an EventArgs structure and an StateEngineEvent structure with about the same content, this class is a StateEngineEvent that can contain information from a .Net event.

Properties

EventArgs	R/W	EventArgs from the .Net event
Sender `object`	R/W	Sender from the .Net event.

Constructors

`public GenericEvent`1(object eventType, object sender, eventArgs);` Initializes a new instance of the GenericEvent class.
eventType — Type of the event.

sender	Sender from the .Net event.
eventArgs	EventArgs from the .Net event.

Methods

7.2.9 Class ImageDescriptionAttribute

```
public class ImageDescriptionAttribute : Attribute
```

Summary
Adds a document image describing this state.
Properties

DefaultImageFileExtension `string`	R	The default image filename extension.
Description `string`	R	A descriptive text for the image.
ImageFileName `string`	R	The file name of the image.
StateMachineDefaultMainImage `string`	R	The default filename of the main image of the StateMachine.

Constructors

```
public ImageDescriptionAttribute();
```
Description of an image with its default file name and no description.

```
public ImageDescriptionAttribute(string imageFileName);
```
Description of an image with its default file name.

imageFileName	The file name of the image. Keep null or empty to use default state image filename.

```
public ImageDescriptionAttribute(string imageFileName, string description);
```
Description of an image given a file name.

imageFileName	The file name of the image. Keep null or empty to use default state image filename.
description	A descriptive text for the image.

Methods

GetDefaultImageFileName
```
public string GetDefaultImageFileName(string factoryName);
```
Gets the default filename (including relative path and extension) given a state.

factoryName	

7.2.10 Class InitialState

```
abstract public class InitialState : PseudoState
```

Summary
Initial state is the first state to be entered in a state machine. There should only be one initial state, and the state engine automatically enters that state at startup. It is a pseudo state so it must direclty transition to another state. The class does not introduce any new behaviour from pseudostate, but marks that it is an intial state.
Constructors

```
protected InitialState();
```

Initializes a new instance of the StateEngine.InitialState class.

7.2.11 Class KeywordDescriptionAttribute

```
public class KeywordDescriptionAttribute : Attribute
```

Summary
Add a keyword to the state. Group the keywords in categories.
Properties

Category `object`	R	Category that the keyword belongs to.
Description `string`	R	Descriptive text.
Keyword `object`	R	Keyword that should be associated with the state.
KeywordText `string`	R	Keyword as a string.

Constructors

`public KeywordDescriptionAttribute(object category, object keyword, string description);` Add a keyword to the state. Group the keywords in categories.	
category	Category that the keyword belongs to.
keyword	Keyword that should be associated with the state.
description	Descriptive text.

`public KeywordDescriptionAttribute(object category, object keyword);` Add a keyword to the state. Group the keywords in categories.	
category	Category that the keyword belongs to.
keyword	Keyword that should be associated with the state.

Methods

CompareTo `public int CompareTo(object obj);` Compares this attribute with another one.	
obj	

7.2.12 Class LogEventArgs

```
public class LogEventArgs : EventArgs
```

Summary
The Log Event args is used to carry log strings when logging internally in the state machine. Applications can hook on the StateMachine.OnLog event in order to catch loggins from the inner workings of the statemachine.
Properties

LogText `string`	R	The log text from the Statemachine.
LogType `Lib.StateEngine.LogType`	R	Category of the log entry.

7.2.13 Class PseudoState

```
abstract public class PseudoState : State
```

Summary
A pseudo state is a state that can be entered, but it must transition directly. The state machine can not stay in this state,but must continue directly. Therefore the Enter, and exit methods are sealed and can not be overriden. Insted, the abstract method CreatePseudoStateTransition method must be overriden. It receives the entry information and must return a transition direclty. The pseudo state can be used to implement choce states, and is the base class for initial states.

Constructors

`protected PseudoState();` Initializes a new instance of the StateEngine.PseudoState class.

Methods

CreatePseudoStateTransition `abstract protected Lib.StateEngine.Transition CreatePseudoStateTransition(Lib.StateEngine.StateEntry stateEntry);` The CreatePseudoStateTransition method must be overriden. It receives the state entry, and must make a new transition directly.	
stateEntry	

Enter `protected void Enter(Lib.StateEngine.StateEntry stateEntry, ref Lib.StateEngine.Transition@ transition);` The Exit method may not be used in a Pseudo state. It must transition directly after enter.	
stateEntry	
transition	

Exit `protected void Exit();` The Exit method may not be used in a Pseudo state. It must transition directly after enter.

HandleEvent `protected void HandleEvent(Lib.StateEngine.StateEngineEvent stateEngineEvent);` The HandleEvent method may not be used in a Pseudo state. It must transition directly after enter.	
stateEngineEvent	

7.2.14 Class State

```
abstract public class State : Object
```

Summary
State is the base state of all states in the state machine.

Properties

Active `bool`	R	Indicates that this is the current active state of the machine.
ApplicationText `string`	R	An additional text that shows up in the visualizer, that for instance can

		be used to point out application specific states.
CreatedByFactory `string`	R/W	Name of the state factory that created the state object.
FactoryName `string`	R	The factory name of the state (the full class name).
InstanceName `string`	R	The name of this particular instance of this state (the hierarchical name of the state, starting with the name of the statemachine, through all parent composite states up to this state).
LogName `string`	R	The name of the state used for logging.
Name `string`	R	The name of the state.
ParentState `Lib.StateEngine.CompositeState`	R	Provides a reference to the composite state this state is contained in. If it is in the root of the state machine, it will be null.
ParentStateMachine `Lib.StateEngine.StateMachine`	R	The parent state machine for the state.
StateType `Lib.StateEngine.StateType`	R	The StateType.

Constructors

```
protected State();
```
Initializes a new instance of the StateEngine.State class.

Methods

ActivateTimer
```
protected void ActivateTimer(Lib.StateEngine.Timer timer);
```
Activates the supplied timer.

timer	The timer to activate.

ClearPendingEvents
```
protected void ClearPendingEvents();
```
Clears all events waiting in the event queues.

Dispose
```
protected void Dispose(bool disposing);
```
Disposes the resources owned by the state object.

disposing	

Dispose
```
public void Dispose();
```
Disposes all the owned resources in the state.

Enter
```
protected void Enter(Lib.StateEngine.StateEntry stateEntry);
```
Enter is called when the state machine enters the state. Override this method to be able to run code at the state Entry.

stateEntry	

Enter
```
protected void Enter(Lib.StateEngine.StateEntry stateEntry,
ref Lib.StateEngine.Transition@ transition);
```
Enter is called when the state machine enters the state. Override this method to be able to run code at the state entry. If a transition should be performed, create a transition object and return it in the transition out property.

stateEntry	Information about the entry of the state.
transition	Out parameter, that should be set to either the reference to a transition object or null.

Exit
```
protected void Exit();
```
Override this method to implement code that should be run at state exit.

HandleEvent
```
protected void HandleEvent(Lib.StateEngine.StateEngineEvent
stateEngineEvent);
```
Override to receive incoming events. If the event is handled, the application must set the event.Handled = true.

stateEngineEvent	The event object that should be handled.

HandleEvent
```
protected void HandleEvent(Lib.StateEngine.StateEngineEvent
stateEngineEvent, ref Lib.StateEngine.Transition@
transition);
```
Override to receive incoming events. If the event is handled, the application must set the event.Handled = true.

stateEngineEvent	The event object that should be handled.
transition	Out parameter that should be set to either the reference to a transition object or null.

RemovePendingEvents
```
protected void
RemovePendingEvents(Lib.StateEngine.StateEngineEventPredicate{``0}
predicate, comparisonObject);
```
Removes all pending event that matches the supplied predicate.

predicate	The predicate that is used to match the event.
comparisonObject	The comparison object that is used in the StateEngineEventPredicate.

RemovePendingEventsOfType
```
protected void RemovePendingEventsOfType(object eventType);
```
Clears all the events in the resend queue that matches the eventType.

eventType	

7.2.15 Class StateChangedEventArgs

```
public class StateChangedEventArgs : EventArgs
```

Summary
Argument in the StateChanged events. It contains the new and the old state in a state transition.
Properties

NewState `Lib.StateEngine.State`	R	The new state that is entered.
OldState `Lib.StateEngine.State`	R	The state that is exited
Transition `Lib.StateEngine.Transition`	R	The Transition.

Constructors

`public StateChangedEventArgs(Lib.StateEngine.State oldState, Lib.StateEngine.State newState, Lib.StateEngine.Transition transition);`
Constructor for the StateChanged Event Arguments

oldState	The State object representing the state that was exited.
newState	The State object representing the state that was entered.
transition	The Transition.

7.2.16 Class StateDescriptionAttribute

`public class StateDescriptionAttribute : Attribute`

Summary
An attribute that can be applied to state classes, that is used to document the state.
Properties

Description `string`	R	Description.
DescriptionType `Lib.StateEngine.StateDescriptionType`	R	The category of the description.

Constructors

`public StateDescriptionAttribute(Lib.StateEngine.StateDescriptionType descriptionType, string description);`
Creates a state engine description for the class.

descriptionType	Category for the description.
description	Description.

`public StateDescriptionAttribute(Lib.StateEngine.StateDescriptionType descriptionType, Type type);`
Creates a state engine description for the class.

descriptionType	Category for the description.
type	Type that stands for the description.

Methods

CompareTo
`public int CompareTo(object obj);`
Compares this attribute with another one.

obj	

7.2.17 Class StateEngineEvent

```
public class StateEngineEvent : Object
```

Summary
Event class that implements the Event interface. It is the base class to be used to send events to states in the state machine.

Properties

Handled `bool`	R/W	Indicates if the event has been handled. If the state is handled, it is the application's responsiblity to set this flag. the State engine never changes it.
Priority `int`	R/W	Specifies the priority of this event. High priority is defined by lower integers. Highest priority is thereby Int.Min. Default priority is 0.
Type `object`	R	Defines the type of the event. Can be any reference or value type (through boxing). Recommended is use of a user-defined enumeration. The EventType is a arbitary type, preferrably enums, but any type can be used.

Constructors

`public StateEngineEvent(object eventType);`
Initializes a new instance of the StateEngine event with the default priority 0.

eventType	Object identifying the event.

`public StateEngineEvent(object eventType, int priority);`
Initializes a new instance of the StateEngine event with the specified priority

eventType	Object identifying the event.
priority	Priority specifier, Lower number specifies that it will be handled sooner than higher numbers. Default priority is 0.

Methods

7.2.18 Class StateEngineException

```
public class StateEngineException : Exception
```

Summary
Wayne.Lib.StateEngine exception is a general exception that can be thrown from the state engine library.

Constructors

`public StateEngineException();`
Initializes a new instance of the StateEngineException class.

`public StateEngineException(string message);`
Initializes a new instance of the StateEngineException class.

message	

`public StateEngineException(string message, Exception inner);`
Initializes a new instance of the StateEngineException class.

message	
inner	

7.2.19 Class StateEntry

```
public class StateEntry : Object
```

Summary
StateEntry contains information about the entry of a state. It is produced by the state transition lookup. The application receives it as an in parameter to the State.Enter() method. What is of intrest to the application might be the sourceTransition, and the attached source Event object.

Properties

HistoryType `Lib.StateEngine.HistoryType`	R	History type that should be used when entering a composite state.
SourceTransition `Lib.StateEngine.Transition`	R	The transition that resulted in the state change.
TargetStateFactoryName `string`	R	The new state that is entered.

Constructors

```
public StateEntry(Lib.StateEngine.Transition
sourceTransition, string targetStateFactoryName,
Lib.StateEngine.HistoryType historyType);
```
Constructor for StateEntry

sourceTransition	The transition that resulted in the state change.
targetStateFactoryName	The new state that is entered.
historyType	History type that should be used when entering a composite state.

7.2.20 Class StateFactories

```
public class StateFactories : Object
```

Summary
A collection of State factories. A state machine can have states created by multiple state factories. When a state object should be created, the factories are queried one by one for the state name.

Constructors

```
public StateFactories();
```
Constructor

Methods

AddFactory
```
public void AddFactory(Lib.StateEngine.IStateFactory
stateFactory);
```
Add a factory to the StateFactory Collection.

stateFactory	The factory that should be added.

Clear
```
public void Clear();
```
Clears the list.

CreateState
```
public Lib.StateEngine.State CreateState(string
stateFactoryName);
```
Calls each registered State Factory to create the requested State. If more than one factory returns a state with the specified name, the method will throw an exception. If no factory returns a state, null will be returned. If exactly one state was created, it will be returned.

stateFactoryName	Name of the State that is going to be created.
Return value	The created state

7.2.21 Class StateMachine

```
abstract public class StateMachine : Object
```

Summary
State machine is the core engine in the Wayne.Lib.StateEngine model. It contains the states, contains the information about the state- transition lookup table, and handles the external and internal events. Normally it should not be neccesary to override this class.

Fields

disposed `bool`	Tells whether the object is disposed or not.
debugLogger `Lib.Log.DebugLogger`	The DebugLogger.
logCategory `object`	The LogCategory.

Properties

CreatedStates `Collections.ObjectModel.ReadOnlyCollection{ Wayne.Lib.StateEngine.State}`	R	List of the created states in the machine.
CurrentState `Lib.StateEngine.State`	R	Current state object.
Depth `int`	R/W	Indicates how many levels the state machine is from the bottom state machine. The first state machine has depth 0. States in a composite state to that machine has depth 1 and so on.
LogNameKind `Lib.StateEngine.StateNameKind`	R/W	The kind of name to use for logging the state name.
Name `string`	R	Name of the state machine given when the state machine was created.
Started `bool`	R	Indicates that the state machine has been started.
StateTransitionLookup `Lib.StateEngine.StateTransitionLookup`	R	The state transition lookup object. Use to set up the state machine in code.

Methods

ActivateTimer
```
public void ActivateTimer(Lib.StateEngine.Timer timer);
```

Activate the supplied timer. This should be called from state object when they want a timer functionality.	
timer	The timer to activate

AddStateFactory
```
public void AddStateFactory(Lib.StateEngine.IStateFactory factory);
```
Adds a custom state factory to the factory collection, so it can be used to create State objects.

factory	The factory to add

ClearStateFactories
```
public void ClearStateFactories();
```
Clears the list of state factories.

Create
```
public Lib.StateEngine.StateMachine Create(string name, Lib.Log.DebugLogger debugLogger, object logCategory);
```
Creates and returns StateMachine object. This method creates Threaded Statemachine. To create other implementations, use the Create(StateMachineType stateMachineType, string name) method instead.

name	Name of the state machine.
debugLogger	The DebugLogger to use.
logCategory	The log category.
Return value	A statemachine object

Create
```
public Lib.StateEngine.StateMachine Create(string name, Lib.StateEngine.StateMachineType stateMachineType, Lib.Log.DebugLogger debugLogger, object logCategory);
```
Creates and returns StateMachine object. The state machine type determines wich implementation should be used.

name	Name of the state machine.
stateMachineType	Type of state machine implementation that should be created.
debugLogger	The DebugLogger to use.
logCategory	The log category.

CreateState
```
familyorassembly Lib.StateEngine.State CreateState(string stateFactoryName);
```
Creates a state from the supplied state name.

stateFactoryName	

Dispose
```
protected void Dispose(bool disposing);
```
The private dispose method taking into account if it is disposed by the finalizer or through a explicit Dispose() call.

disposing	

Dispose
`public void Dispose();`
Disposes resources.

Finalize
`protected void Finalize();`
Destructor (finalizer). Stops the state machine if it has not already been done.

IncomingEvent
`abstract public void IncomingEvent(Lib.StateEngine.StateEngineEvent stateEngineEvent);`
Sends an event into the state machine.

stateEngineEvent	Incoming event

Initialize
`public void Initialize();`
Initializes the state machine, creates all state objects.

RemovePendingEvents
`public void RemovePendingEvents(Lib.StateEngine.StateEngineEventPredicate{``0} predicate, comparisonObject);`
Removes all pending event that matches the supplied predicate.

predicate	The predicate that is used to match the event.
comparisonObject	The comparison object that is used in the StateEngineEventPredicate.

RemovePendingEventsOfType
`public void RemovePendingEventsOfType(object eventType);`
Removes all pending events that matches the specified event type. Comparison is made with the .Equals method.

eventType	

Start
`abstract public void Start();`
Starts the state machine. This method should be called after the machine has been configured and equipped with at least one state factory.

Events

OnFinalStateEntered
`public EventHandler OnFinalStateEntered;`
Even that is fired when the final state of a state machine is reached.

OnStateChange
`public EventHandler{Wayne.Lib.StateEngine.StateChangedEventArgs} OnStateChange;`
Event that signals just before a state change will occur.

OnStateChanged
`public EventHandler{Wayne.Lib.StateEngine.StateChangedEventArgs}`

`OnStateChanged;` Event that signals just after a state change has occured.

7.2.22 Class StateTransitionLookup

```
public class StateTransitionLookup : Object
```

Summary
StateTransitionLookup is used by the State machine to find the state to change to when a transition occurs. It uses a dataset that contains the actual data.

Constructors

`public StateTransitionLookup(Lib.StateEngine.StateMachine stateMachine);` Constructor	
stateMachine	

Methods

AddTransition `public void AddTransition(object transitionType);` Adds a transition from state of class TFromState to the class TToState, when the transitiontype is inserted.	
transitionType	Transition type.

AddTransition `public void AddTransition(object transitionType, Lib.StateEngine.HistoryType historyType);` Adds a transition from state of class TFromState to the class TToState, when the transitiontype is inserted.	
transitionType	Transition type.
historyType	

AddTransition `public void AddTransition(string fromStateFactoryName, string toStateFactoryName, object transitionType);` Adds a transition from state of class fromStateFactoryName to the class toStateFactoryName, when the transitiontype is inserted.	
fromStateFactoryName	String containig the factory name for the fromState.
toStateFactoryName	String containig the factory name for the toState.
transitionType	Transition type.

Dispose `public void Dispose();` Disposes the resources owned by the StateTransition Lookup.

Finalize `protected void Finalize();` Finalizer

GetAnyStateTransitionNameList `public String[] GetAnyStateTransitionNameList();` Returns a list of the AnyState-transition names (no duplicates).

GetNextState `public Lib.StateEngine.StateEntry GetNextState(string sourceStateFactoryName, Lib.StateEngine.Transition transition);` Performs a Lookup for the next state when in the source state, and are going to perform the specified transition	
sourceStateFactoryName	The state the state machine is in when performing transition
transition	The requested transition.
Return value	Returns the next state that should be entered based on the transition.

GetNextStateName `public string GetNextStateName(string sourceStateFactoryName, object transitionType, Lib.StateEngine.HistoryType@ historyType, Boolean@ fromAnyState);`	
sourceStateFactoryName	
transitionType	
historyType	
fromAnyState	

GetSourceStateNameList `public String[] GetSourceStateNameList();` Returns a list of the source State names that is in the lookup table.

GetStateNameList `public String[] GetStateNameList();` Returns a list of state factory names that is in the lookup table.

GetStateTransitionNameList `public String[] GetStateTransitionNameList(string sourceStateFactoryName);` Returns a list of the transition names from a state (no duplicates).	
sourceStateFactoryName	

GetStateTransitionsDict `public Collections.Generic.Dictionary{System.String,System.String} GetStateTransitionsDict(string sourceStateFactoryName, bool includeAnyStates);` Returns a dictionary of the transitions from a state (dictionary key) to a state (dictionary value).	
sourceStateFactoryName	
includeAnyStates	

GetTransitionInfoArray `public Lib.StateEngine.TransitionInfo[] GetTransitionInfoArray(bool includeAnyStates);` Gets a list of all transitions.

includeAnyStates	Should the AnyState-transitions be included?

RemoveTransition
`public void RemoveTransition(object transitionType);`
Removes a transition from state of class TFromState to the class TToState of a certain transitionType.

transitionType	Transition type.

RemoveTransition
`public void RemoveTransition(object transitionType);`
Removes a transition from state of class TFromState of a certain transitionType.

transitionType	Transition type.

RemoveTransition
`public void RemoveTransition();`
Removes all transitions from state of class TFromState to the class TToState.

RemoveTransition
`public void RemoveTransition(string fromStateFactoryName, string toStateFactoryName, object transitionType);`
Removes a transition from the state fromStateFactoryName to the state toStateFactoryName of a certain transitionType.

fromStateFactoryName	String containig the factory name for the fromState.
toStateFactoryName	String containig the factory name for the toState.
transitionType	Transition type.

RemoveTransition
`public void RemoveTransition(string fromStateFactoryName, object transitionType);`
Removes a transition from the state fromStateFactoryName of a certain transitionType.

fromStateFactoryName	String containig the factory name for the fromState.
transitionType	Transition type.

RemoveTransition
`public void RemoveTransition(string fromStateFactoryName, string toStateFactoryName);`
Removes all transitions from the state fromStateFactoryName to the state toStateFactoryName.

fromStateFactoryName	String containig the factory name for the fromState.
toStateFactoryName	String containig the factory name for the toState.

RemoveTransitions
`public void RemoveTransitions();`
Removes all transition from state of class TFromState.

RemoveTransitions
`public void RemoveTransitions(string fromStateFactoryName);`
Removes all transition from the state fromStateFactoryName.

fromStateFactoryName	String containig the factory name for the fromState.

ReplaceTransition `public void ReplaceTransition(object transitionType);` Replaces the existing transition from a state and the given TransitionType, to another state.	
transitionType	Transition type.

ReplaceTransition `public void ReplaceTransition(object transitionType, Lib.StateEngine.HistoryType historyType);` Replaces the existing transition from a state and the given TransitionType, to another state.	
transitionType	Transition type.
historyType	

ReplaceTransition `public void ReplaceTransition(string fromStateFactoryName, string toStateFactoryName, object transitionType);` Replaces the existing transition from a state and the given TransitionType, to another state.	
fromStateFactoryName	String containig the factory name for the fromState.
toStateFactoryName	String containig the factory name for the toState.
transitionType	Transition type.

7.2.23 Class TimeoutDescription

```
public class TimeoutDescription : EventDescriptionAttribute
```

Summary
Describe the timeout / transition relationship for a state class of the generic TimeoutState.
Constructors

`public TimeoutDescription(string conditionText, object transitionType);` Describe the timeout-event / transition relationship for a state class.	
conditionText	A descriptive text for the condition.
transitionType	Transition that is performed.

7.2.24 Class Timer

```
public class Timer : Object
```

Summary
Timer is a timer that should be used within the state machine.
Properties

ClearAtStateExit `bool`	R/W	Specifies if the timer should be disabled automatically when the owning state is exit.
Enabled `bool`	R	If the timer is active.
EventType `object`	R/W	Event type that should be set in the TimerEvent when it fires.
IsPeriodic	R/W	Specifies if the timer should fire several times.

`bool`		
OwnerState `Lib.StateEngine.State`	R/W	State object that created the timer.
UserToken `object`	R	User-supplied token that is sent in to the constructor at creation time.

Constructors

`public Timer(Lib.StateEngine.State ownerState, object eventType, int interval, object userToken);`
Constructor for Timer

ownerState	State that created the timer
eventType	Type that should be set in the timer event that is sent when the timer fires
interval	Interval of the timer
userToken	Token object that is supplied by the user and that is returned in the TimerEvent.

Methods

Disable
`public void Disable();`
Disables the timer.

Dispose
`public void Dispose();`
Disposes the timer.

Enable
`public void Enable();`
Enables the timer

Finalize
`protected void Finalize();`
Finalizer

7.2.25 Class TimerEvent

`public class TimerEvent : StateEngineEvent`

Summary
Event class for the Timer events.

Properties

UserToken `object`	R	The UserToken of the Timer.

7.2.26 Class Transition

`public class Transition : Object`

Summary
The transition class is used by state objects to signal that a change has occured. The transition is interpreted by the statemachine's State-transition lookup table. it determines which the next state should be. The transition type identifies the transition and is supplied by the application. It is recommended that enums are used as transition types, but any type can be used.

Properties

Name `string`	R	Represents the type name. It returns the Type.ToString(). This is used by the statemachine to find the transition in the lookup table.
Sender `Lib.StateEngine.State`	R	The state that issued the transition.
SourceEvent `Lib.StateEngine.StateEngineEvent`	R	If the transition is generated when a state has received an event, it can be supplied in this property.
Type `object`	R/W	Type of the transistion.

Constructors

`public Transition(Lib.StateEngine.State sender, object type);`
Transition constructor. assumes that the Source Event is null.

sender	State that issued the transition
type	Type of the transition

Methods

GetTransitionName
`public string GetTransitionName(object transitionType);`
Returns the name of the transition to use when persisting configuration etc.

transitionType	

7.2.27 Class TransitionInfo

`public class TransitionInfo : Object`

Summary
A class wrapping information about a transition.

Properties

FromStateFactoryName `string`	R	The factory name of the "From" state.
HistoryType `Lib.StateEngine.HistoryType`	R	The History Type.
ToStateFactoryName `string`	R	The factory name of the "To" state.
TransitionName `string`	R	The name of the transition.

Constructors

`public TransitionInfo(string fromStateFactoryName, string transitionName, string toStateFactoryName, Lib.StateEngine.HistoryType historyType);`
Constructor.

fromStateFactoryName	
transitionName	
toStateFactoryName	
historyType	

Methods

7.3 Enumerations

7.3.1 Enumeration BasicTransitionType

Summary
Basic Transition Types is for automatic transitions.
Fields

Init	Init transition is used when entering the initial state in the state machine.
Done	When a composite state has reached a final state, the state machine automatically fires a Done transition in the state machine that owns the composite state.
Error	This is a transition that is sent by the state machine either if an internal error is detected or if an exception has slipped out of the user code in the State.Enter, Exit, HandleEvent methods. Add a state transition configuration to handle this transition in each state machine.
Timeout	Generic Timeout transition.

7.3.2 Enumeration HistoryType

Summary
History type defines the way that a composite state is entered.
Fields

None	Only the initial state is entered. No history is recalled
Shallow	If the composite has been active before, the state that was active last is entered. Shallow means that entering the recalled last state but when entering that state, it is done with no history.
Deep	If a composite state has been active before, the state that was active last is entered, Deep means that if the recalled state is a composite it will also be entered with deep history.
Explicit	Explicit history type is *only* used when issuing explicit transitions. If a state machine is configured with this history type, an error will be thrown.

7.3.3 Enumeration LogType

Summary
Categorizes the log entries in the OnLog event from the State machine.
Fields

Enter	Logging when a state entry is performed.
Exit	Logging when a state exit is performed.
HandleEvent	Logging when an event is sent into a state for handling.
Error	Logging when an exception has been unhandled in the user code.
Debug	Misc. debug logging.
UnhandledTransition	Warnings about unhandled transitions.

7.3.4 Enumeration StateDescriptionType

Summary
Used in the DescriptionAttribute to categorize the different descriptions.
Fields

Summary	A summary of the state behaviour.

Enter	A desciptive text about the state entry.
Exit	A desciptive text about the state exit.s
Timeout	If the state has a timeout behaviour, it is described in this category.

7.3.5 Enumeration StateMachineType

Summary
Enumeration of the types of statemachines that can be created. Used in the factory method StateMachine.Create.
Fields

Threaded	A statemachine that runs in a separate thread.
Synchronous	A state machine that uses the thread that calls IncomingEvent method to process the state changes and the state code execution.

7.3.6 Enumeration StateNameKind

Summary
The two kinds of names for a state.
Fields

FactoryName	The factory name of a state (the full class name).
InstanceName	The name of this particular instance of a state (the hierarchical name of the state, starting with the name of the statemachine, through all parent composite states up to the state itself).

7.3.7 Enumeration StateType

Summary
The state types as an enumeration. This can be used in a future design tool.
Fields

InitialState	Intial state
PseudoState	Pseudo state
State	Ordinary state
CompositeState	Composite state
FinalState	Final state

7.4 Delegates

StateEngineEventPredicate`1

```
public StateEngineEventPredicate`1
(Lib.StateEngine.StateEngineEvent stateEngineEvent,
compareObject);
```

Delegate that is used to match StateEngine events.

stateEngineEvent	The event that should be evaluated
compareObject	The object that should be used in the comparison.
Return value	True if the stateengineevent matches the predicate, otherwise false.

8 Namespace Wayne.Lib.StateEngine.Description

Classes

AsyncDoneDescription	Describe the AsyncDone / transition relationship for a state class of the generic AsyncWorkState.

8.1 Classes

8.1.1 Class AsyncDoneDescription

```
public class AsyncDoneDescription : EventDescriptionAttribute
```

Summary
Describe the AsyncDone / transition relationship for a state class of the generic AsyncWorkState.
Constructors

`public AsyncDoneDescription(string conditionText, object transitionType);` Describe the AsyncDone-event / transition relationship for a state class.	
conditionText	A descriptive text for the condition.
transitionType	Transition that is performed.

9 Namespace Wayne.Lib.StateEngine.Generic

Interfaces

IGenericState`1	Common interface for all the generic state classes. It is used in the state factory code.

Classes

AsyncWorkState`1	Generic state that enables descendant classes to execute code on a worker thread when the state is active. When the processing is completed, i.e. the PerformWork returns, the state will post a transition of the type specified in the constructor. When exitting the state, we for the worker thread to complete before continuing. Descendant classes can also use the AbortWork() method to signal to the worker thread that it should exit as fast as possible. The PerformWork method should periodically check the Aborted property and if that is true, exit as fast as possible.
CompositeState`1	Generic composite state class that has a main object of a generic type.
FinalState`1	Generic final state class that has a main object that is the user's choice.
InitialState`1	Generic initial state class that has a main object of a generic type.
PseudoState`1	Generic pseudo state class that has a main object of a generic type.
State`1	Generic state class that has a main object of a generic type.
TimeoutState`1	Generic timeout state class that has a main object of a generic type.

Enumerations

GenericEventType	Built-in event type definition that is used by the timeoutstate timer handling.

9.1 Interfaces

9.1.1 Interface IGenericState`1

```
public interface IGenericState`1
```

Summary
Common interface for all the generic state classes. It is used in the state factory code.
Properties

WritableMain	R/W	Writable main object. Only used in the state factory code.

9.2 Classes

9.2.1 Class AsyncWorkState`1

```
abstract public class AsyncWorkState`1 : State<TMain>
```

Summary
Generic state that enables descendant classes to execute code on a worker thread when the state is active. When the processing is completed, i.e. the PerformWork returns, the state will post a transition of the type specified in the constructor. When exitting the state, we for the worker thread to complete before continuing. Descendant classes can also use the AbortWork() method to signal to the worker thread that it should exit as fast as possible. The PerformWork method should periodically check the Aborted property and if that is true, exit as fast as possible.

Properties

Aborted `bool`	R	Signals that the PerformWork method should return as fast as possible.

Constructors

`public AsyncWorkState`1(object doneTransitionType);`
Initializes a new instance of the AsyncWorkState class.

doneTransitionType	

Methods

AbortWork
`protected void AbortWork();`
Signals to the thread that it should exit as quick as possible. Can be overridden by descendant classes to create user code to abort.

Enter
```
protected void Enter(Lib.StateEngine.StateEntry stateEntry,
ref Lib.StateEngine.Transition@ transition);
```
Called when entering the state.

stateEntry	
transition	

Exit
`protected void Exit();`
Called when exitting the state.

HandleEvent
```
protected void HandleEvent(Lib.StateEngine.StateEngineEvent
stateEngineEvent, ref Lib.StateEngine.Transition@
transition);
```
Handles events.

stateEngineEvent	
transition	

PerformWork
`abstract protected void PerformWork();`
Method that is called in a worker thread.

WorkDone
```
protected void WorkDone(ref Lib.StateEngine.Transition@
transition);
```
Method that is called when the work is done. Override to specify another transition type than the one specified in the constructor.

transition	

9.2.2 Class CompositeState`1

```
abstract public class CompositeState`1 : CompositeState
```

Summary
Generic composite state class that has a main object of a generic type.
Properties

Main	R	The main object.

Constructors

```
protected CompositeState`1();
```
Initializes a new instance of the StateEngine.Generic.CompositeState`1 class.

9.2.3 Class FinalState`1

```
abstract public class FinalState`1 : FinalState
```

Summary
Generic final state class that has a main object that is the user's choice.
Properties

Main	R	The main object.

Constructors

```
protected FinalState`1();
```
Initializes a new instance of the StateEngine.Generic.FinalState`1 class.

9.2.4 Class InitialState`1

```
abstract public class InitialState`1 : InitialState
```

Summary
Generic initial state class that has a main object of a generic type.
Properties

Main	R	The main object.

Constructors

```
protected InitialState`1();
```
Initializes a new instance of the StateEngine.Generic.InitialState`1 class.

9.2.5 Class PseudoState`1

```
abstract public class PseudoState`1 : PseudoState
```

Summary
Generic pseudo state class that has a main object of a generic type.
Properties

Main	R	The main object.

Constructors

```
protected PseudoState`1();
```
Initializes a new instance of the StateEngine.Generic.PseudoState`1 class.

9.2.6 Class State`1

```
abstract public class State`1 : State
```

Summary
Generic state class that has a main object of a generic type.

Properties

Main	R	The main object

Constructors

```
protected State`1();
```
Initializes a new instance of the StateEngine.Generic.State`1 class.

9.2.7 Class TimeoutState`1

```
abstract public class TimeoutState`1 : State<TMain>
```

Summary
Generic timeout state class that has a main object of a generic type.

Properties

TimeoutInterval `int`	R	Method that is used by descendant classes to set the timeout of the state.

Constructors

```
protected TimeoutState`1();
```
Initializes a new instance of the StateEngine.Generic.TimeoutState`1 class.

Methods

CancelTimer
```
protected void CancelTimer();
```
Cancels the currently running timer.

Enter
```
protected void Enter(Lib.StateEngine.StateEntry stateEntry,
ref Lib.StateEngine.Transition@ transition);
```
See State.Enter

stateEntry	
transition	

Exit
```
protected void Exit();
```
See State.Exit

HandleEvent
```
protected void HandleEvent(Lib.StateEngine.StateEngineEvent
stateEngineEvent, ref Lib.StateEngine.Transition@
transition);
```
HandleEvent is sealed, use HandleNonTimeoutEvent method to override instead.

stateEngineEvent	
transition	

HandleNonTimeoutEvent
```
abstract protected void
HandleNonTimeoutEvent(Lib.StateEngine.StateEngineEvent
stateEngineEvent, ref Lib.StateEngine.Transition@
transition);
```
Method that is implemented by descendant classes to receive events that were not the timeout event.

stateEngineEvent	

transition	

ResetTimer
```
protected void ResetTimer();
```
Restarts the timer. If a timer is active, it is restarted, and if the timer is not active anymore, it is started again.

Timeout
```
abstract protected void Timeout(ref
Lib.StateEngine.Transition@ transition);
```
Method that is used by descendant classes to be signaled when the timeout has fired.

transition	

9.3 Enumerations

9.3.1 Enumeration GenericEventType

Summary
Built-in event type definition that is used by the timeoutstate timer handling.
Fields

Timeout	Timeout event.
AsyncDone	Asynchronous processing completed.