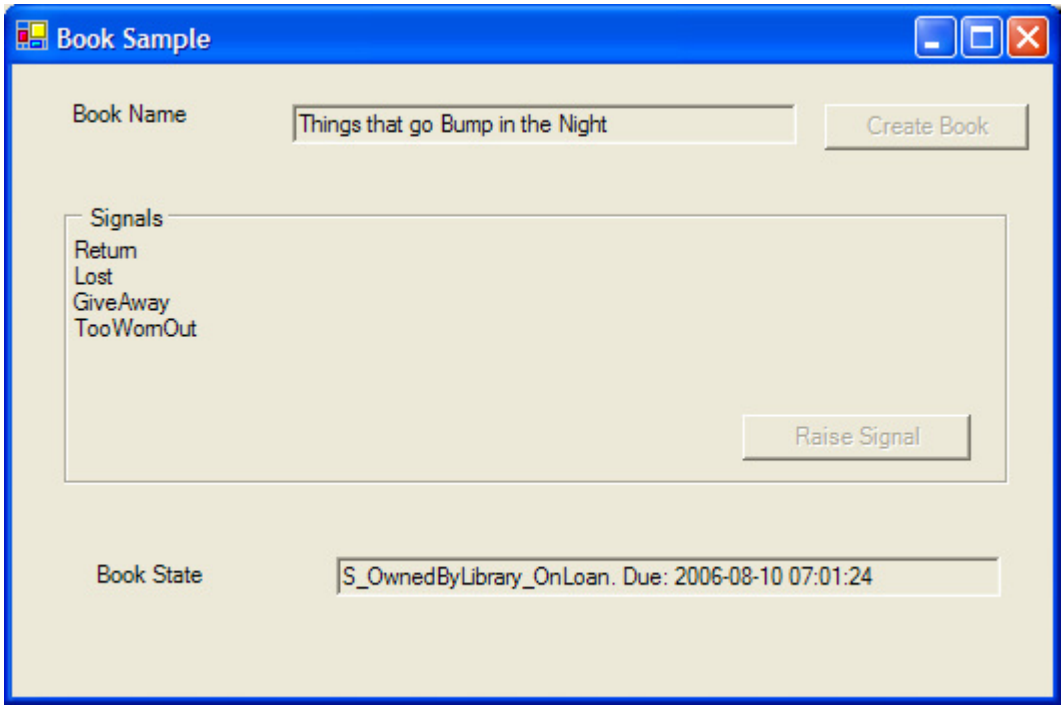
ateProto – Saving and Restoring the State Machine



Introduction

We have thus far covered three topics about stateproto and the qf4net extension framework that it uses.

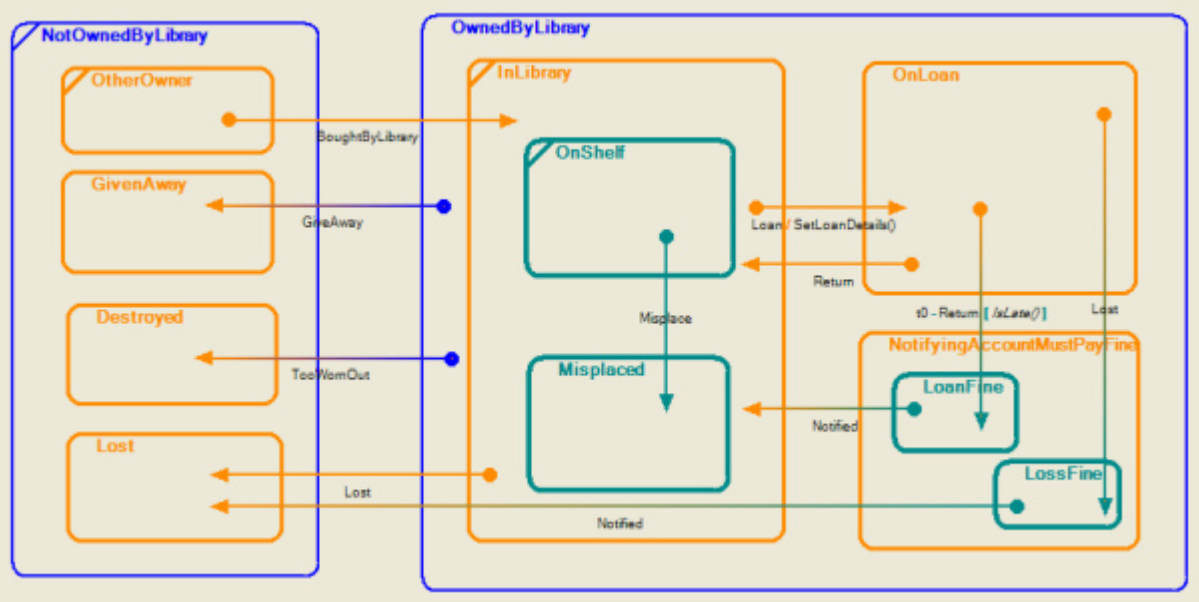
- **The Basic StateMachine**
    - Using a watch as a sample
- Showing how to hook up state change events for logging.
- Explains the basic use of StateProto as a state modelling tool
- **Running multiple instances**
    - Still using the watch as a sample
- Shows how to create and activate multiple instances
- Introduces three threading models
- Shows simple animation of the StateProto diagrams
- **Getting a few state machine based objects to interact**
    - This sample introduces the elements in lighting a lighter
- Introduces ports as a linking and message transfer mechanism between state machines
- Uses the threading ideas from the preceding article to execute multiple objects (as state machines)

In this article I will cover the topic of reloading a state machine so that it is possible to get it back to the state it was in before you shutdown (or after a crash!). I plan to keep this sample as simple as possible by looking at a single state machine. The mechanism presented is generalisable to multiple state machines and a framework can be built to automate the storage mechanics.

The Sample

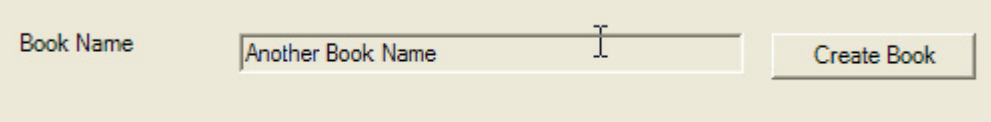
Todays sample is a simple workflow for the lifecycle of a library book. This lifecycle can be broken down into the following main components:

1. the book exists but is not owned by the library
2. its owned by the library and is in a good enough condition to still put on loan
3. is on loan and return cycle for many many times (hopefully it is not lost too soon)
4. the book finally is either
    - so worn out that it is retired, or
5. it is lost
6. finally,if either overdue or lost while on loan then a fine needs to be payed.
   I do not show a user account that will be updated with this fine amount in this sample.



Book Hierarchical State Machine

The sample application is very simple. The aim is really to show how to save an instance of a state machine (the book hsm here) and later restore it.

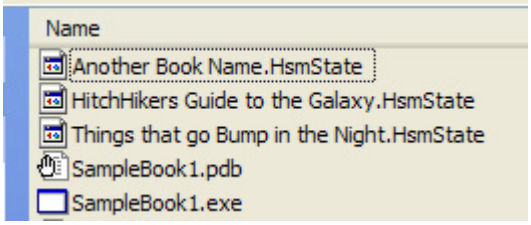


Enter a Book Name and then click "Create Book"

The sample starts up with a default book name - but can be changed to any other name. Then by clicking "Create Book" you will get two effects

1. The code creates a book state machine.
2. A file is saved with an extension .HsmState.

Now, if you were to restart the application at this point and put in the same book name - you would have no proof that the state machine has been restored (other than by looking in the console).



State can be saved to file via Binary Serialisation of the Memento

So you will see that the default start state is "NotOwnedByLibrary". The only signal available from "NotOwnedByLibrary" is "BoughtByLibrary" (I will explain how the known signals are known later on). Select "BoughtByLibrary" from the ListBox and click "Raise Signal" (which should become enabled). The hsm will move into "OwnedByLibrary_InLibrary_OnShelf" state. Restarting the application at this point and entering the same book name will restore the hsm to "OwnedByLibrary_InLibrary_OnShelf".

Also note that if you do take a book on loan that the due date is one minute later :-).

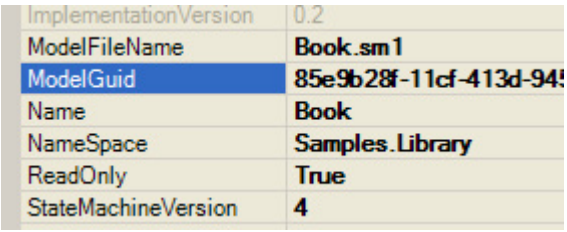
Finally, when extracting the sample, please note that:

- the code is in hsm\samples\SampleSaveAndRestore,
- the demo executable is in hsm\bin,
- and that hsm\stateProto.exe is the state machine design application. You can load the hsm from hsm\samples\SampleSaveAndRestore\doc\Book.sm1.
  Please refer to the first state proto article on how this works.

Memento - the key to saving and restoring an hsm

A state machine generally indicates the lifecycle of a single class. Often states tend to replace multiple boolean variables (or test cases) that would otherwise be taken up by different variables. However, there is a cost to pay - as it might become important to store the state a state machine is in. This state consists of the following components:

- The **CurrentState** of the workflow/machine.
- Any **extended state** held by instance variables (fields) of the state machine. This includes data like a book's due date.
- If the state machine has **history transitions** - then information about history states will need to be extracted from the state machine.
- Finally, these extra fields are needed for LQHsm-based state machines:
  - **Id** - a unique identity applied to each state machine
- **GroupId** - concurrently related state machines will share a group id.
- And these are needed for correlation with the stateProto source diagram model:
  - **ModelVersion** - Indicates the diagram source model version that this state machine was generated from.
- **ModelGuid** - Indicates the diagram source model identification code that this state machine was generated from.
  These are not necessarily common to most state machine models.



ModelVersion and ModelGuid are part of the StateProto model information - not really QHsm

If you can imagine that there is some way to expose all this data without making it directly available then we could save this data to database or file and later restore an hsm by reloading the data from the store. Unfortunately, direct access to the state machine's fields would mean that every field must be exposed as a public property (ignoring private reflection). This is especially the requirement if I wanted to build some generic mechanism for saving state machines. I said unfortunately because public exposure of every field (even as a property) would violate encapsulation and allow possibly uncontrolled access to the field/property data outside of the thread of control of the state machine. It would be even worse if this exposure was unintended by the original state machine designer.

A better way to extract this information is to use some form of the `Memento` pattern. You can find a complete explanation of this pattern on Wikipedia. Memento provides a separate class whose primary role is to act as a transport of internal data for another class and has no other behaviour. This means that we can store the state machine's internal information into a separate memento object. The memento class we use is called `LQHsmMemento`. `LQHsmMemento` is a type specific memento class for `LQHsm` based state machines. It exposes properties and methods that allows a state machine derived off `LQHsm` to externalise its current data state.

How this happens is as follows:

```
1       // Two steps to get to a state machine's internal data state:
2       // 1. Create a memento
3       ILQHsmMemento memento = new LQHsmMemento ();
4       // 2. Ask the state machine to save its data state to the newly created memento.
5       Hsm.SaveToMemento (memento);
```

`SaveToMemento` is a composed method that starts off by saving the simple bits (currentState, id, guid, etc) and then calls other methods to any save history state information and another to save additional fields. State machines can carry additional fields. These fields are referred to as the state machine's extended state.

Unfortunately, even though most of this process is automated - there is still one step that the hsm developer needs to do. There are two methods on `LQHsm` state machines that must be overriden (assuming the state machine does have some extended state variables):

- `SaveFields(ILQHsmMemento memento)`

  Within this method call `memento.AddField(...)` for every data field that needs to be saved. These fields would have been declared within the state machine by the developer and are not code generated fields.

- `RestoreFields(ILQHsmMemento memento)`

  In here call `memento.GetFieldFor(...)` for every data field that needs to be restored. `GetFieldFor()` expects that the field has been previously saved to the memento.

```
1       #region Save/Restore
2       protected override void SaveFields(ILQHsmMemento memento)
3       {
4           memento.AddField ("DueDate", _DueDate, typeof(DateTime));
5       }
6
7       protected override void RestoreFields(ILQHsmMemento memento)
8       {
9           _DueDate = (DateTime)memento.GetFieldFor ("DueDate").Value;
10      }
11      #endregion
```

LQHsm memento definition

Here is the full definition for the interface for a LQHsmMemento:

```
1       public interface ILQHsmMemento
2       {
3           string Id { get; set; }
4           string GroupId { get; set; }
5           string ModelVersion { get; set; }
6           string ModelGuid { get; set; }
7           MethodInfo CurrentStateMethod { get; set; }
8           void ClearHistoryStates ();
9           void AddHistoryState (string name, MethodInfo state);
10          void ClearFields ();
11          void AddField (string name, object value, Type type);
12
13          IStateMethodInfo GetHistoryStateFor (string name);
```

```
14          IFieldInfo GetFieldFor (string name);
15
16          IStateMethodInfo[] GetHistoryStates ();
17          IFieldInfo[] GetFields ();
18      }
```

Concurrency

While `hsm.SaveToMemento(...)` can be called at any time - we must remember that the state machine is in general running from a different thread than the one that you would be calling `SaveToMemento(...)` from. This means that when you call `SaveToMemento(...)` it might be that the state machine is at that point processing an event which could result in a transition. The way to get around this is to create a class that implements `IQSimpleCommand` and pass it on to `hsm.EventManager.AsyncDispatch(cmd)`. What this does is pass a command onto the same queue that the state machine is executing its events from in the order of submission (i.e. FIFO order). The end result is that the command runs as soon as the state machine has completed all preceding event processing. Remember from the previous articles that the EventManager represents both the thread of execution of a state machine and its event queue where events can be signals and timeouts.
Be careful to also note the there is an unfortunate overload of the use of the term "**event**". Events in state machine terminology are also sometimes referred to as *messages* or just plain *signals*. This is unlike a C# event which represents a delegate sink interface for notifications (a.k.a also called *events*) *(sigh)*.

```
1       public interface IQSimpleCommand
2       {
3           void Execute ();
4       }
```

The sample provides two commands:

- SaveCmd
- RestoreCmd

`SaveCmd` is created and dispatched every time a signal is processed. It could also have been initiated only after every state change. On the other hand `RestoreCmd` is created and dispatched only when the state machine is created and also only if a file is found that contains a saved instance of this state machine. Notice than if the state machine is being restored then `init()` must not be called.

Saving

Saving is done after some notification that the state machine has done work. This can be done by trapping the `PolledEvent` event on the eventManager belonging to the state machine. This is done within the `BookFrame` class in the sample. Remember that a polledEvent is notified every time an event is pulled off the eventManager's event queue. There are two notifications - one before the event is processed and one after.

```
1       public class BookFrame
2       {
3           // ...
4           void RegisterStateChange(ILQHsm hsm)
5           {
6               hsm.EventManager.PolledEvent += new
PolledEventHandler(EventManager_PolledEvent);
7           }
8           // ...
9           private void EventManager_PolledEvent(IQEventManager eventManager, IQHsm hsm,
IQEvent ev, PollContext pollContext)
10          {
11              if(pollContext == PollContext.AfterHandled && hsm == _Book)
12              {
13                  SaveHsmToFile ();
14                  DoStateChange ((ILQHsm)hsm);
15              }
```

```
        }
    }
```

Now every time the state machine handles an event it will call `SaveHsmToFile()` which sends the `SaveCmd` for execution and then binary serialises the memento to file.

You might be wondering why I did not rather hook into the state change event. I could have with one provisor - which is that I only do something when the change is an entry type (i.e. `StateLogType.Entry`) and that I use the `LogStateEventArgs.State` property to establish my current state as the state machine will not yet have set its current state when this notification is raised. I mention using only the entry type as an optimisation so as not to be saving too often. Notice that I do a similar check within the `eventManager.PolledEvent` handler to ensure that this polledEvent is for my hsm.

```
    private void SaveHsmToFile()
    {
        // not really necessary to create a new instance every time
        SaveCmd cmd = new SaveCmd (_Book, _StorageFileName);
        // callback to this frame to save the memento to file
        // could also have done the writing to file within the SaveCmd
        cmd.Completed += new HsmMementoCompleted(cmd_SaveCompleted);
        // send the cmd for execution
        _Book.EventManager.AsyncDispatch(cmd);
    }

    private void cmd_SaveCompleted(IQSimpleCommand command, ILQHsmMemento memento)
    {
        SaveCmd cmd = (SaveCmd) command;
        // standard binary serialisation to file.
        using(StreamWriter sw = new StreamWriter(cmd.FileName))
        {
            BinaryFormatter bf = new BinaryFormatter();
            bf.Serialize(sw.BaseStream, memento);
        }
    }
```

The `SaveCmd` simply calls `SaveToMemento()` in its `Execute` method.

```
    public override void Execute()
    {
        // create empty memento
        ILQHsmMemento memento = new LQHsmMemento ();
        // get the state machine to fill the memento with its
        // internal information
        Hsm.SaveToMemento (memento);
        // tell the frame
        DoCompleted (memento);
    }
```

Restoring

Restoring is not much different from Saving. The only difference is that there is only one point where it is required to do a restore which is after creating the state machine but before calling `Init()`. Note that `Init()` is not called if the state machine is restored.

```
    // Restore from within the Frame
    public BookFrame(){
        // ...

        // Init or Restore
        if(File.Exists(_StorageFileName))
        {
            RestoreHsmFromFile ();
        }
        else
        {
            Init ();
            SaveHsmToFile ();
        }
```

```
15      }
16
17      //...
18
19      private void RestoreHsmFromFile()
20      {
21          using(StreamReader sr = new StreamReader(_StorageFileName))
22          {
23              // deserialise memento
24              BinaryFormatter bf = new BinaryFormatter();
25              object obj = bf.Deserialize(sr.BaseStream);
26              ILQHsmMemento memento = (ILQHsmMemento) obj;
27              // dispatch restore cmd to other thread
28              RestoreCmd cmd = new RestoreCmd (_Book, memento);
29              cmd.Completed += new HsmMementoCompleted(cmd_RestoreCompleted);
30              _Book.EventManager.AsyncDispatch (cmd);
31          }
32      }
```

The command executes...

```
1       public override void Execute()
2       {
3           Hsm.RestoreFromMemento (_Memento);
4           DoCompleted (_Memento);
5       }
```

One Last Thing

Filling in the ListBox

You will notice that every time the state machine changes state that the ListBox of available signals changes to show all possible available signals. I must admit that although I can code generate this list automatically - I have not as yet done so. Instead what I used was a mechanism I added in a while back to provide me with a list of string values as attributes against the current state which I called `StateCommand` attributes.

```
1       [StateCommand ("Loan")]
2       [StateCommand ("Lost")]
3       protected virtual QState S_OwnedByLibrary_InLibrary (IQEvent ev){
```

These attributes can be any string value but for this sample I made them the same as the actual signals that can be sent by the specified state. Using this I can get all attributes of the current state and any of its parents and create a signal list. Note that this code is currently in the Book hsm directly and should be moved into the LQHsm base class.

Also note that in general these commands are open to the interpretation and use for the state machine implementor and need not be the same as the signals responded to by the state.

The state commands can be filled in against any state in StateProto

Commands entry dialog in StateProto

Summary

Saving and restoring a state machine is not too complicated. There are overheads involved in doing so - and this will need to be examined in more detail in future. But the idea is simple - call `SaveToMemento()` to save and `RestoreFromMemento()` to restore.

On the plus side - this can be converted into a generic persistance service storing to file or maybe even a sqlite or mssql database.

The Book sample is extremely simplistic - but I hope that it succeeds in making the concept clear.

Coming Up

1. There are so many things. Maybe an article on the way I do state machine design. Any requests would be welcome.

History

Fourth article for stateProto beta release showing how state machines can be saved and restored.

References

Miro Samek's Quantum Hierarchical State Machine technology can be found at http://www.quantum-leaps.com/

Dr Rainer Hessmer's QF4Net port can be found at http://www.hessmer.org/dev/qhsm/ and at sourceforge at http://sourceforge.net/projects/qf4net/ which Dr Hessmer agreed for me to publish.

StateProto can be found at http://sourceforge.net/projects/stateproto/

Miro Samek explains QHsm samek0311.pdf

CSharp Code Formatter at http://www.manoli.net/csharpformat/

Memento at Wikipedia

First Article: StateProto Beta - State Chart Designer for Qf4Net

Second Article: StateProto – Executing Multiple StateMachines

Third Article: StateProto – Interacting State Machines