SSE 661

Software Architecture

Project #4

by

Jason Payne

June 26, 2015

TABLE OF CONTENTS

Topics Covered	Topic Examples
System Architecture Design	• Active Object

1. Active Object

The Active Object design pattern decouples method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control.

1.1 Step-by-Step Design

The AO design pattern will be implemented for a system that places items for sale as auctions. To be fair to all bidders, the system must support multiple clients bidding on an item at the same time (concurrently). So, to support concurrency, the bid items will be implemented as active objects. ***Figure 1-1*** reveals the class diagram for the system.

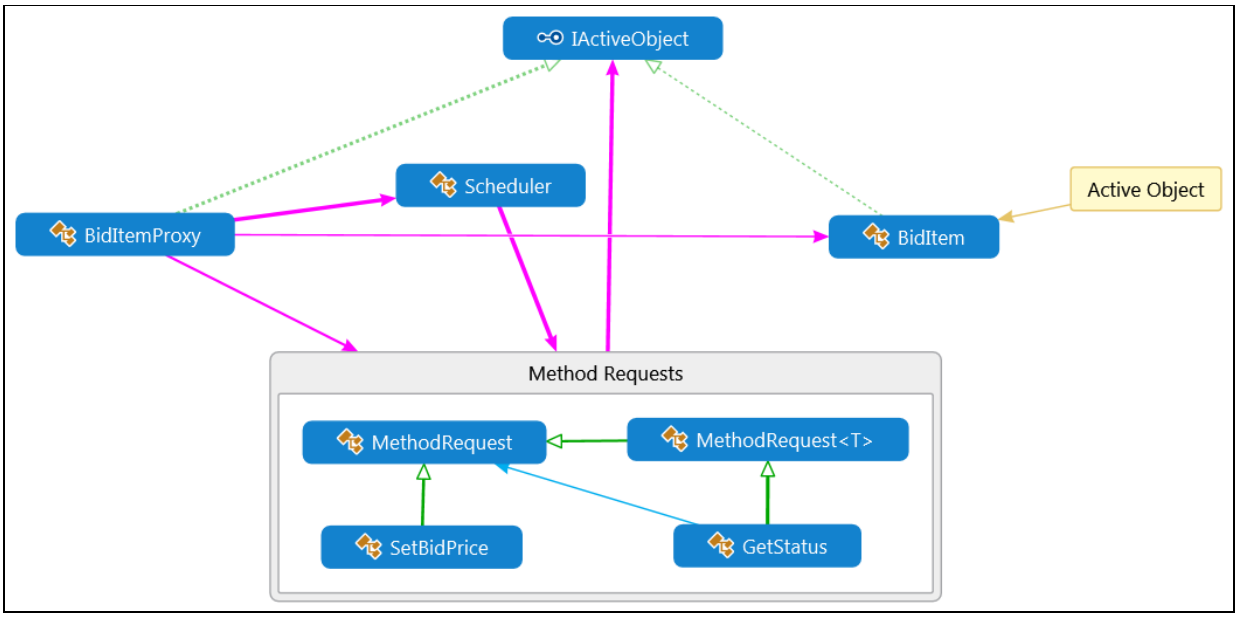

Figure 1-1: *Class diagram of demonstration project's active object pattern*

1. **Implement the Servant (active object).**
   For this project, the Servant defines the behavior and state of an item that is presented for sale on a bidding website (BidItem). Since the Servant must allow for concurrent access to its methods and properties, this will be the Active Object for the system. An interface is created (IActiveObject) so that the Proxy object for this class will have a contract that enforces the same behavior definition of the Servant onto the Proxy.

2. **Implement the Proxy.**
   With the Servant defined and implemented, the Proxy (BidItemProxy) is created by implementing the `IActiveObject` interface. This ensures that the same behavior defined by the Servant will be the same behavior defined by this Proxy object.

3. **Implement the Method Requests.**
   Two base classes are created for concrete method request classes to inherit. MethodRequest defines the interface that `void` methods should inherit and

MethodRequest<T> defines the interface that value-returning methods should inherit. The value returned is in the form of a "future" (or `Task<T>`) so that the client knows that a value will eventually be returned at some point in the future.

`MethodRequest`-based classes (see the MethodRequest Classes in Appendix A) are created for each method specified in the `IActiveObject` interface. These classes provide a way for the Proxy to communicate method calls with the Scheduler and Servant.

4. **Implement the Activation List.**
   The activation list was not explicitly implemented for this system due to the fact that the .NET Framework 4.5+ provides several thread-safe collections that serve well as the role of the activation list. For this system, the BlockingCollection<T> was used as a thread-safe queue that blocks its thread (the background thread in this case, not the client thread) when attempting to dequeue an object and the queue is empty.
5. **Implement the Scheduler.**
   Determining the implementation of the Scheduler is dependent upon the priority of method requests and the state of active object at the time that the method request is ready to be dequeued. This can change the design due to the fact that a queue-like behavior may not provide the necessary capabilities to properly order the method requests (i.e. objects in a queue cannot be accessed by index (or priority) so an array-based collection may be necessary). For this project, however, the guard method(s) always returned true and the priority was always a First-In-First-Out (FIFO) order so a thread-safe queue was used.
6. **Determine rendezvous and return value policy.**
   The `Task<T>` class provided by the .NET Framework was used to implement an asynchronous rendezvous and return value policy. This allowed calls made to the Servant (through the Proxy) to behave as non-blocking calls (asynchronous) on the client thread. And with the `Task<T>` class, there are attributes that allow the call to be made as a blocking call, exceptions to be handled explicitly, and the status of the task (or thread) to be monitored. The simulation runs presented at the end of Appendix A demonstrate the policy.

1.2 System Analysis

The benefits of utilizing the AO pattern for this application include:

- **Introduces and enhances application concurrency.** The actions of clients on bid items run asynchronous to the execution of those actions.
- **Simplifies complexity of synchronization** between objects. All of the synchronization code is encapsulated within a single class, the Scheduler, which is solely responsible for starting the background thread used to execute the method requests and ordering the requests based on synchronization constraints (none for this particular system).
- **Transparently leverages available parallelism.** When run on a virtual machine with limited thread and processing capabilities, the simulations run almost synchronously.

However, when run on a machine that has multiple cores and threads, the system automatically scales to leverage the capabilities without requiring any changes to code.

- **Method invocation is decoupled from method execution.** Because the method execution can differ from the order of method invocation, the system's performance and flexibility is improved when compared to synchronous implementation that causes blocking calls on the client thread.

The risks of utilizing the AO pattern for this application include:

- **Classes required per operation/attribute** defined in the active object. Because this system (and pattern) uses request classes for each property or method, the number of classes in the system could get large.
- **Negative performance overhead.** The switching and synchronization between threads can cause a significant use of system resources negatively impacting overall performance.
- **Limited debugging capabilities**. By implementing in C# using Visual Studio, the Visual Studio IDE provides debugging features for asynchronous applications. However, implementing in another language (and IDE) could severely limit and increase the difficulties of debugging the application because of a lack of asynchronous debugging support.

Overall, the AO Pattern fits the needs of this system perfectly and lends itself to simple scalability and flexibility. For this reason, this pattern is a great fit for this system and is definitely recommended!

Appendix A – *Active Object Implementation*

This section provides the source code for the system created for exercising the Active Object design pattern.

IActiveObject

This is the interface of the active object (servant). It represents the bid item that clients can place bids on and receives statuses about.

```
/// <summary>
/// Interface of the active object.
/// </summary>
public interface IActiveObject
{
    void SetBidPrice(string bidder, double bid);

    Task<string> GetStatus(string bidder);
}
```

BidItem

This class represents the concrete implementation of the active object (servant). For the purposes of this project, it allows clients (through the proxy) to place bids on items up for sale and to receive the statuses of the bid item. The `GetStatus` method returns a `Task<string>` object which serves the same purpose as a "future" object as presented in the text.

```
/// <summary>
/// Implementation of the active object.
/// </summary>
public class BidItem : IActiveObject
{
    public int ProductId { get; private set; }

    public double Price { get; private set; }

    public string WinningBidder { get; private set; }

    private BidItem()
    {
        Price = 0;
        WinningBidder = string.Empty;
    }

    public BidItem(int id) : this()
    {
        ProductId = id;
    }

    public void SetBidPrice(string bidder, double bid)
    {
        Console.WriteLine("{0} is attempting to set bid price to ${1}.", bidder, bid);

        if (bid > Price)
        {
            Console.WriteLine("{0} is the high bidder!", bidder);
            Price = bid;
```

```
            WinningBidder = bidder;
            return;
        }

        Console.WriteLine("Bid of ${0} has been rejected because it is lower than " +
                          "the highest bid.", bid);
    }

    public Task<string> GetStatus(string bidder)
    {
        Func<string> result =
            () =>
            {
                Console.WriteLine("Retrieving status for {0}...please wait", bidder);
                return (bidder.Equals(WinningBidder))
                            ? string.Format("{0} is the highest bidder!", bidder)
                            : string.Format("{0} has been outbid.", bidder);
            };
        return new Task<string>(result);
    }
}
```

Method Request Interfaces (MethodRequest and MethodRequest<T>)

These classes provide the interface that all method request objects must implement. A generic and non-generic base class is provided to accommodate different return types of methods on the active object. Methods that return `void` will inherit from the `MethodRequest` base class and those that have an actual return type will inherit from the `MethodRequest<T>` so that a future object can automatically be setup for such objects.

```
/// <summary>
/// Base class that all active object method requests must inherit from.  Method
/// requests that do not return a value (future) should inherit from this class
/// specifically.
/// </summary>
public abstract class MethodRequest
{
    protected IActiveObject _bidItem;

    protected MethodRequest()
    {
    }

    protected MethodRequest(IActiveObject bidItem)
    {
        _bidItem = bidItem;
    }

    /// <summary>
    /// Guard method to be utilized by the method request dispatcher.
    /// </summary>
    public virtual bool CanExecute
    {
        get { return true; }
    }
```

```
    /// <summary>
    /// This method is called by the dispatcher (from the active object's context
    /// thread) to perform the actual execution of the requested method made by the
    /// client.  Inheriting method request classes will override this with the call to
    /// the active object's method that it represents.
    /// </summary>
    public abstract void Execute();
}
```

```
/// <summary>
/// Generic base class that all value-returning method request classes should inherit
/// from.
/// </summary>
/// <typeparam name="T">The type of value (future) that should be returned by this
/// method request.</typeparam>
public abstract class MethodRequest<T> : MethodRequest
{
    /// <summary>
    /// The future to be returned by the method request.
    /// </summary>
    protected Task<T> _future;

    protected MethodRequest(IActiveObject bidItem) : base(bidItem)
    {
    }
}
```

Scheduler

This class is the method request scheduler that queues/dequeues `MethodRequest` objects for the purposes of allowing concurrent invocations of methods on the active object (`BidItem`). This is accomplished by utilizing a thread-safe collection class (provided by the .NET Framework) as the activation list for the system. The `BlockingCollection` type defaults to a queue-like behavior and automatically supports a producer/consumer concurrency model. Method requests are added to the collection in the client's context thread, but are removed and executed in the background "listener" loop that is in the active object's context thread.

```
/// <summary>
/// The method request scheduler that is responsible for tracking requested
/// invocations of active object methods and executing the requests on the active
/// object in a separate thread (dispatch thread).
/// </summary>
public class Scheduler
{
    /// <summary>
    /// Asynchronous activation container that manages the method requests in the form
    /// of a queue.
    /// </summary>
    private BlockingCollection<MethodRequest> Queue { get; set; }

    public Scheduler()
    {
        // BlockingCollection accepts a parameter as a capacity which limits the size
        // of the Queue to protect against many service requests which would result in
        // a large Queue negatively affecting performance.  For this project, however,
```

```
        // a simplistic approach is used in which no capacity is set on the Queue.
        Queue = new BlockingCollection<MethodRequest>();

        // Create and start the background dispatch thread.  The dispatch thread
        // starts listening for method requests while the client thread returns to
        // its expected behavior.
        Task.Factory.StartNew(Dispatch);
    }

    /// <summary>
    /// Adds a method request to the activation queue.
    /// </summary>
    /// <param name="methodRequest">The method request to add.</param>
    public void Insert(MethodRequest methodRequest)
    {
        Queue.Add(methodRequest);
    }

    /// <summary>
    /// Listener loop started by the scheduler in a background thread that "listens"
    /// for the existence of method requests.
    /// </summary>
    public void Dispatch()
    {
        do
        {
            // Block the dispatch thread until a request is placed on the Queue.
            var request = Queue.Take();

            // DESIGN DECISION: No guard methods are in place since CanExecute always
            // returns true.

            // Execute the request
            request.Execute();
        } while (true);
    }
}
```

Method Request Classes (SetBidPrice and GetStatus)

These are the concrete implementations of the `MethodRequest` interface and represent the methods that are provided by the active object (e.g. `IActiveObject::SetBidPrice(…)` corresponds to the `SetBidPrice` method request class defined here, etc.). It should be noted that the `SetBidPrice` class inherits from the `MethodRequest` base class while the `GetStatus` class inherits from the `MethodRequest<T>` base class since it is to return a value back to the client thread.

```
/// <summary>
/// Method request class for the IActiveObject.SetBidPrice(...) method.
/// </summary>
public class SetBidPrice : MethodRequest
{
    // Method request parameters
    private readonly string _bidder;
    private readonly double _bid;
```

```
    public SetBidPrice(IActiveObject bidItem, string bidder, double bid) : base(bidItem)
    {
        _bidder = bidder;
        _bid = bid;
    }

    /// <summary>
    /// Called from the dispatch thread to execute the IActiveObject.SetBidPrice(...)
    /// method.
    /// </summary>
    public override void Execute()
    {
        _bidItem.SetBidPrice(_bidder, _bid);
    }
}
```

```
/// <summary>
/// Method request class for the IActiveObject.GetStatus(...) method.
/// </summary>
public class GetStatus : MethodRequest<string>
{
    // Method request parameter
    private readonly string _bidder;

    public GetStatus(IActiveObject bidItem, string bidder, out Task<string> future)
        : base(bidItem)
    {
        _bidder = bidder;
        _future = _bidItem.GetStatus(_bidder);
        future = _future;
    }

    /// <summary>
    /// Called from the dispatch thread to execute the IActiveObject.GetStatus(...)
    /// method.
    /// </summary>
    public override void Execute()
    {
        _future.Start();
    }
}
```

BidItemProxy

This class is the proxy object for the active object. It allows multiple clients to access the active object concurrently. It inherits from the `IActiveObject` interface so that it will properly mimic the attributes and operations that are provided by the actual active object class (`BidItem`). This is also where the `BidItem` and `Scheduler` objects are constructed. The thing to note with this is that when the `Scheduler` is initialized, it automatically starts its listener loop in a background thread.

```
/// <summary>
/// Proxy for the active object class.  It inherits from the same interface as that
/// of the actual active object.
/// </summary>
public class BidItemProxy : IActiveObject
{
```

```
    /// <summary>
    /// Active object that the proxy is "wrapping".
    /// </summary>
    private IActiveObject BidItem { get; set; }

    /// <summary>
    /// Method request scheduler for the active object.
    /// </summary>
    private Scheduler Scheduler { get; set; }

    private BidItemProxy()
    {
        Scheduler = new Scheduler();
    }

    public BidItemProxy(int id) : this()
    {
        BidItem = new BidItem(id);
    }

    /// <summary>
    /// An action method of the active object that is provided by the proxy
    /// </summary>
    public void SetBidPrice(string bidder, double bid)
    {
        Scheduler.Insert(new SetBidPrice(BidItem, bidder, bid));
    }

    /// <summary>
    /// A method of the proxy that returns a "future" result.  This result will be
    /// generated when the method is actually invoked on the active object by the
    /// dispatch thread.  This call is non-blocking.
    /// </summary>
    /// <param name="bidder">Name of the bidder requesting the status.</param>
    /// <returns>A non-blocking future value that will contain the active object's
    /// status.</returns>
    public Task<string> GetStatus(string bidder)
    {
        Task<string> future;
        var request = new GetStatus(BidItem, bidder, out future);
        Scheduler.Insert(request);
        return future;
    }
}
```

Program

This class is the main entry point for the console application. It initializes the proxy class (`BidItemProxy`) and a couple of clients (`Bidder` objects). It then creates two actions that represent client bidding behaviors on a single `BidItem` object. The two actions are then started concurrently using .NET's `Parallel.Invoke` method.

```
internal class Program
{
    private static void Main(string[] args)
    {
        Log.Initialize();
```

```
        Run();
        Console.ReadKey();
        Log.Close();
    }

    private static void Run()
    {
        var bidItemProxy = new BidItemProxy(123);

        var blueBidder = new Bidder("Blue");
        var redBidder = new Bidder("Red");

        Action blueBidAction = () =>
                               {
                                   // Blue bidder places a bid and retrieves the status.
                                   var blue = blueBidder.Name;
                                   bidItemProxy.SetBidPrice(blue, 1);
                                   var blueResult = bidItemProxy.GetStatus(blue);
                                   Console.WriteLine(blueResult.Result);
                               };

        Action redBidAction = () =>
                               {
                                   // Red bidder places a bid and retrieves the status.
                                   var red = redBidder.Name;
                                   bidItemProxy.SetBidPrice(red, 2);
                                   var redResult = bidItemProxy.GetStatus(red);
                                   Console.WriteLine(redResult.Result);
                               };

        // Simulate two bidders bidding on the same bid item (active object).
        Parallel.Invoke(blueBidAction, redBidAction);
    }

    internal class Bidder
    {
        public string Name { get; private set; }

        private Bidder()
        {
        }

        public Bidder(string name)
        {
            Name = name;
        }
    }
}
```

Output:

The simulation was executed four consecutive times which yielded the results below. Note that it appears that while the tasks were performed concurrently, the underlying task scheduling mechanism of the operating system combined with the implementation of the Active Object pattern resulted in different behaviors. However, even with the different timing of the tasks, the results were still correct due to the ability of the system to handle method invocations by serially scheduling the tasks in a thread-safe queue. Had this simulation remained the same and the system made to handle tasks synchronously, the behavior would have been erratic and prone to errors.

Sim Run #1

```
Red is attempting to set bid price to $2.
Red is the high bidder!
Blue is attempting to set bid price to $1.
Bid of $1 has been rejected because it is lower than the highest bid.
Retrieving status for Blue...please wait
Retrieving status for Red...please wait
Red is the highest bidder!
Blue has been outbid.
```

Sim Run #2

```
Blue is attempting to set bid price to $1.
Blue is the high bidder!
Red is attempting to set bid price to $2.
Red is the high bidder!
Retrieving status for Red...please wait
Retrieving status for Blue...please wait
Blue has been outbid.
Red is the highest bidder!
```

Sim Run #3

```
Blue is attempting to set bid price to $1.
Blue is the high bidder!
Red is attempting to set bid price to $2.
Red is the high bidder!
Retrieving status for Blue...please wait
Retrieving status for Red...please wait
Red is the highest bidder!
Blue has been outbid.
```

Sim Run #4

```
Blue is attempting to set bid price to $1.
Blue is the high bidder!
Red is attempting to set bid price to $2.
Red is the high bidder!
Retrieving status for Blue...please wait
Retrieving status for Red...please wait
Blue has been outbid.
Red is the highest bidder!
```

Non-Direct Activity Report – Jason Payne

Date	Duration (minutes)	Specific Task / Activity
Sum for Report #1	**1570**	**/ 900 (1 week @ 900/wk)**
Sum for Report #2	**1991**	**/ 900 (1 week @ 900/wk)**
Sum for Report #3	**1927**	**/ 900 (1 week @ 900/wk)**
24-May-2015	81	Research for project #4
7-Jun-2015	210	Research for project #4.
15-Jun-2015	15	Team discussion about project #4.
20-Jun-2015	51	Setup & research for project #4.
21-Jun-2015	420	Research for project #4.
22-Jun-2015	300	Research for project #4.
23-Jun-2015	300	Research for project #4.
24-Jun-2015	225	Research for project #4.
25-Jun-2015	100	Research for project #4.
26-Jun-2015	150	Research for project #4.
Sum for Report #4	**1852**	**/ 1800 (2 weeks @ 900/wk)**
Sum for Class	**7340**	**/ 4500 (5 weeks @ 900/wk)**