Microsoft Logo

Gray Pipe

search clear

Asynchronous Design Pattern Overview

.NET Framework 1.1

One of the innovations provided by the asynchronous pattern is that the caller decides whether a particular call should be asynchronous. It is not necessary for a called object to do additional programming for supporting asynchronous behavior by its clients; asynchronous delegates provide for this in the pattern. The common

language runtime handles the difference between the caller and called object views. The called object can choose to explicitly support asynchronous behavior, either because it can implement it more efficiently than a general architecture or it wants to support only asynchronous behavior by its callers. However, it is recommended that such called objects follow the asynchronous design pattern for exposing asynchronous operations.
Type safety is another innovation for the asynchronous pattern. For asynchronous delegates in particular, a language compiler that targets the .NET Framework and common language runtime can type safe method signatures for the begin and end operations that map to the regular **Invoke** method, for example, for **BeginInvoke** and **EndInvoke**. This is significant because the compiler breaks the synchronous call into the begin and end operations for the asynchronous delegate, and makes it possible to pass only valid parameters.
The basic ideas behind this pattern are as follows:

6. The caller decides whether a particular call should be asynchronous.
7. The called object to do additional programming for supporting asynchronous behavior by its clients, but this is not necessary. The common language runtime infrastructure should be able to handle the difference between the caller and called object views.
8. The called object can choose to explicitly support asynchronous behavior, either because it can implement it more efficiently than a general architecture or it wants to support only asynchronous behavior by its callers. However, it is recommended that such called objects follow the asynchronous design pattern for exposing asynchronous operations.
9. The compiler generates type safe method signatures for **BeginInvoke** and **EndInvoke**, for asynchronous delegates.
10. The .NET Framework provides services needed for supporting an asynchronous programming model. An example partial list of such services is:

    - Synchronization primitives, such as monitors and reader writer locks.
    - Threads and thread pools.
    - Synchronization constructs, such as containers that support waiting on objects.

- Exposure to the underlying infrastructure pieces, such as **IMessage** objects and thread pools.

The pattern breaks down a synchronous call into constituent parts: a begin operation, end operation, and result object. Consider the following example where the **Factorize** method has the potential to take a significant amount of time to complete.

```
public class PrimeFactorizer
{
  public bool Factorize(int factorizableNum, ref int primefactor1, ref int primefactor2)
  {
    // Determine whether factorizableNum is prime.
    // If it is prime, return true. Otherwise, return false.
    // If it is prime, place factors in primefactor1 and primefactor2.
  }
}
```

If the asynchronous pattern is followed, the class library writer adds **BeginFactorize** and **EndFactorize** methods that break the synchronous operation into two asynchronous operations:

```
public class PrimeFactorizer
{
  public bool Factorize(
int factorizableNum,
ref int primefactor1,
ref int primefactor2)
  {
    // Determine whether factorizableNum is prime.
    // if it is prime, return true; otherwise return false.
    // if it is prime, place factors in primefactor1 and primefactor2
  }

  public IAsyncResult BeginFactorize(
int factorizableNum,
ref int primefactor1,
```

```
ref int primefactor2,
AsyncCallback callback,
Object state)
  {
    // Begin factoring asynchronously, and return a result object,
  }

  public bool EndFactorize(
ref int primefactor1,
ref int primefactor2,
IAsyncResult asyncResult
)
  {
    // End (or complete) the factorizing,
    // return the results,
    // and obtain the prime factors.
  }
}
```

The server splits asynchronous operation into its two logical parts: the part that takes input from the client and calls the asynchronous operation, and the part that supplies results of the asynchronous operation to the client.

In addition to the input needed for the asynchronous operation, the first part also takes an **AsyncCallback** delegate to be called when the asynchronous operation is completed. The first part returns a waitable object that implements the **IAsyncResult** interface used by the client to determine the status of the asynchronous operation.

The server also uses the waitable object it returned to the client to maintain any state associated with asynchronous operation. The client uses the second part to obtain the results of the asynchronous operation by supplying the waitable object.

Options available to client for initiating asynchronous operations:

3. Supply the callback delegate when beginning the asynchronous call.

```
public class Driver1
{
```

```
  public PrimeFactorizer primeFactorizer;

  public void Results(IAsyncResult asyncResult)
  {
    int primefactor1=0;
    int primefactor2=0;

    bool prime = primeFactorizer.EndFactorize(
ref primefactor1,
ref primefactor2,
asyncResult);
  }

  public void Work()
  {
    int factorizableNum=1000589023,
    int primefactor1=0;
    int primefactor2=0;
    Object state = new Object();

    primeFactorizer = new PrimeFactorizer();
    AsyncCallback callback = new Callback(this.Results);
    IAsyncResult asyncResult = primeFactorizer.BeginFactorize(
factorizableNum,
ref primefactor1,
ref primefactor2,
callback,
state);
  }
}
```

4. Do not supply the callback delegate when beginning asynchronous operation.

```
public class Driver2
{
  public static void Work()
  {
    int factorizableNum=1000589023,
    int primefactor1=0;
    int primefactor2=0;
    Object state = new Object();

    PrimeFactorizer primeFactorizer = new PrimeFactorizer();
    AsyncCallback callback = new Callback(this.Results);
    IAsyncResult asyncResult = primeFactorizer.BeginFactorize(
factorizableNum,
ref primefactor1,
ref primefactor2,
callback,
state);

    bool prime = primeFactorizer.EndFactorize(
ref primefactor1,
ref primefactor2,
asyncResult);
  }
}
```

See Also

Asynchronous Method Signatures | IAsyncResult Interface | AsyncCallback Delegate for Asynchronous Operations | Asynchronous Programming

Show: Inherited Protected

Dev centers

- Windows
- Office
- Visual Studio
- Microsoft Azure
- More...

Learning resources

- Microsoft Virtual Academy
- Channel 9
- MSDN Magazine

Community

- Forums
- Blogs
- Codeplex

Support

- Self support

Programs

- BizSpark (for startups)
- Microsoft Imagine (for students)

United States (English)

- Newsletter
- Privacy & cookies
- Terms of use
- Trademarks

logo

© 2017 Microsoft