**Mediator Pattern** – Objects know about a mediator object but not about the objects attached to the mediator. It allows an object to change its state. The mediator then raises an event that notifies the listening objects to change state.

Singleton class – one instance of the class in memory for a given application so all objects can share the same instance. To do it we create a static read-only field that will return the instance of the mediator to be shared across the components and we make the constructor private so only the class can create a new instance. Once an object gets an instance of the mediator class they can subscribe to the OnChanged event of the mediator and be notified of changes.

So if an object wants to subscribe to the mediator it should:

Mediator.GetInstance().JobChanged += (s, e) => { do what you want to do on event raised};

The subscribing object adds its own lambda to the event which will be executed when the event is raised. The invocation of the event is in the OnEvent method which will automatically invoke the list of attached delegates. The delegates themselves can execute any code as long as the method signature matches the delegate definition of the event.

Advantages: Loosely coupled objects. The listeners don’t have to know about each other or the event raiser.

Aynchronous delegates – delegates raised and called on a separate thread.

Delegate.BeginInvoke() – spawns a separate thread to execute the event handler.

This.BeginInvoke(delegate, theArgs) – calls the delegate to execute on the owning thread. Must be passed a delegate and the args.

InvokeRequired – determines if the calling thread has ownership of the object.

To view active threads – Debug->windows->Threads

BackgroundWorker – a built in component that can achieve the same thing as asynchronous delegates. It’s in the windows forms toolbox.

BackGroundWorker.RunWorkerAsync() – raises an event that calls the event handler DoWork of the BackgroundWorker on a back ground thread.

BackGroundWorker.ReportProgress() raises ProgressChanged event

BackGroundWorker.CancelAsync() – sets CancelPending to true so you can halt DoWork execution in the worker thread and will raise the RunWorkerCompleted event.

BackgroundWorker has 3 events:

DoWork - run on background thread

ProgressChanged - run on main thread

RunWorkerCompleted - run on main thread

This means we don’t need to re-route data and component manipulation from the background thread to the main thread.

Starting threads:

Threads take a special type of delegate called StartThread which defines a pipeline for a handler with no params.

Var t = new Thread( new ThreadStart(StartProcessing));

Public void StartProcessing(){}

t.Start();

still can’t change main thread components on the background thread created by the above

Have to query InvokeRequired and then invoke a delegate to recursive execute on the same method but from the main thread.