How do I effectively design my application where most classes depend on ILogger?

Asked 12 years, 9 months ago Modified 1 year, 8 months ago Viewed 23k times

74

I am injecting a `Logger` component into *all* my classes. Most of my classes have the `Logger` property defined, except where there is inheritance chain (in that case only the base class has this property, and all the deriving classes use that). When these are instantiated through Windsor container, they would get my implementation of `ILogger` injected into them. I'm currently using Property Injection, because injecting everything into the constructor doesn't feel good.

Can I pull them from the container as they are transient in nature? Do I have to register them with the container and inject in the constructor of the needing class? Also just for one class I don't want to create a `TypedFactory` and inject the factory into the needing class.

Another thought that came to me was `new` them up on need basis. So if I new them up, I will have to manually instantiate the `Logger` in those classes. How can I continue to use the container for ALL of my classes?

Example Windsor registrations:

```
//Install QueueMonitor as Singleton
Container.Register(Component.For<QueueMonitor>().LifestyleSingleton());
//Install DataProcessor as Trnsient
Container.Register(Component.For<DataProcessor>().LifestyleTransient());

Container.Register(Component.For<Data>().LifestyleScoped());
```

Example classes:

```
public class QueueMonitor
{
    private dataProcessor;

    public ILogger Logger { get; set; }

    public void OnDataReceived(Data data)
    {
        // pull the dataProcessor from factory
        dataProcessor.ProcessData(data);
    }
}

public class DataProcessor
{
```

```
    public ILogger Logger { get; set; }

    public Record[] ProcessData(Data data)
    {
        // Data can have multiple Records
        // Loop through the data and create new set of Records
        // Is this the correct way to create new records?
        // How do I use container here and avoid "new"
        Record record = new Record(/*using the data */);
        ...
        // return a list of Records
    }
}

public class Record
{
    public ILogger Logger { get; set; }

    private _recordNumber;
    private _recordOwner;

    public string GetDescription()
    {
        Logger.LogDebug("log something");
        // return the custom description
    }
}
```

Questions:

1. How do I create new `Record` object without using "new"?
2. `QueueMonitor` is `Singleton`, whereas `Data` is "Scoped". How can I inject `Data` into `OnDataReceived()` method?

.net **dependency-injection** **castle-windsor** **ioc-container**

Share

Improve this question

Follow

edited Apr 4, 2023 at 14:05

Steven

172k ● 25 ● 346 ● 447

asked Mar 27, 2012 at 14:59

user1178376

978 ● 2 ● 11 ● 24

@Steven I added a code sample showing the Logger usage. Do you think this is a bad design? – user1178376 Mar 27, 2012 at 15:27

1 Can you illustrate with concrete code, even better a test, what you're trying to achieve? – Mauricio Scheffer Mar 27, 2012 at 17:46

Sorry for poorly constructing my question. I added more code to explain my case. – user1178376 Mar 27, 2012 at 19:26

1 Answer

Sorted by: Highest score (default)

314

From the samples you give it is hard to be very specific, but in general, when you inject `ILogger` instances into most services, you should ask yourself two things:

1. Do I log too much?
2. Do I violate the SOLID principles?

1. Do I log too much

You are logging too much, when you have a lot of code like this:

```
try
{
   // some operations here.
}
catch (Exception ex)
{
    this.logger.Log(ex);
    throw;
}
```

Writing code like this comes from the concern of losing error information. Duplicating these kinds of try-catch blocks all over the place, however, doesn't help. Even worse, I often see developers log and continue by removing the last `throw` statement:

```
try
{
   // some operations here.
}
catch (Exception ex)
{
    this.logger.Log(ex); // <!-- No more throw. Execution will continue.
}
```

This is most of the cases a bad idea (and smells like the old VB `ON ERROR RESUME NEXT` behavior), because in most situations you simply have not enough information to determine whether it is safe continue. Often there is a bug in the code or a hiccup in an external resource like a database that caused the operation to fail. To continue means that the user often gets the idea that the operation succeeded, while it hasn't. Ask yourself: what is worse, showing users a generic error message saying that there something gone wrong and ask them to try again, or silently skipping the error and letting users *think* their request was successfully processed?

Think about how users will feel if they found out two weeks later that their order was never shipped. You'd probably lose a customer. Or worse, a patient's MRSA registration silently fails, causing the patient not to be quarantined by nursing and

resulting in the contamination of other patients, causing high costs or perhaps even death.

Most of these kinds of try-catch-log lines should be removed and you should simply let the exception bubble up the call stack.

Shouldn't you log? You absolutely should! But if you can, define one try-catch block at the top of the application. With ASP.NET, you can implement the `Application_Error` event, register an `HttpModule` or define a custom error page that does the logging. With Win Forms the solution is different, but the concept stays the same: Define one single top most catch-all.

Sometimes, however, you still want to catch and log a certain type of exception. A system I worked on in the past let the business layer throw `ValidationExceptions`, which would be caught by the presentation layer. Those exceptions contained validation information for display to the user. Since those exceptions would get caught and processed in the presentation layer, they would not bubble up to the top most part of the application and didn't end up in the application's catch-all code. Still I wanted to log this information, just to find out how often the user entered invalid information and to find out whether the validations were triggered for the right reason. So this was no error logging; just logging. I wrote the following code to do this:

```
try
{
   // some operations here.
}
catch (ValidationException ex)
{
    this.logger.Log(ex);
    throw;
}
```

Looks familiar? Yes, looks exactly the same as the previous code snippet, with the difference that I only caught `ValidationException` exceptions. However, there was another difference that can't be seen by just looking at this snippet. There was only *one place* in the application that contained that code! It was a decorator, which brings me to the next question you should ask yourself:

2. Do I violate the SOLID principles?

Things like logging, auditing, and security, are called cross-cutting concerns (or aspects). They are called *cross cutting*, because they can cut across many parts of your application and must often be applied to many classes in the system. However, when you find you're writing code for their use in many classes in the system, you are most likely violating the SOLID principles. Take for instance the following example:

```
public void MoveCustomer(int customerId, Address newAddress)
{
    var watch = Stopwatch.StartNew();

    // Real operation

    this.logger.Log("MoveCustomer executed in " +
        watch.ElapsedMiliseconds + " ms.");
}
```

Here you measure the time it takes to execute the `MoveCustomer` operation and you log that information. It is very likely that other operations in the system need this same cross-cutting concern. You start adding code like this for your `ShipOrder`, `CancelOrder`, `CancelShipping`, and other use cases, and this leads to a lot of code duplication and eventually a maintenance nightmare (I've been there.)

The problem with this code can be traced back to a violation of the SOLID principles. The SOLID principles are a set of object-oriented design principles that help you in defining flexible and maintainable (object-oriented) software. The `MoveCustomer` example violated at least two of those rules:

1. The Single Responsibility Principle (SRP)—classes should have a single responsibility. The class holding the `MoveCustomer` method, however, does not only contain the core business logic, but also measures the time it takes to do the operation. In other words, it has multiple *responsibilities*.
2. The Open-Closed principle (OCP)—it prescribes an application design that prevents you from having to make sweeping changes throughout the code base; or, in the vocabulary of the OCP, a class should be open for extension, but closed for modification. In case you need to add exception handling (a third responsibility) to the `MoveCustomer` use case, you (again) have to alter the `MoveCustomer` method. But not only do you have to alter the `MoveCustomer` method, but many other methods as well, as they will typically require that same exception handling, making this a sweeping change.

The solution to this problem is to extract the logging into its own class and allow that class to wrap the original class:

```
// The real thing
public class MoveCustomerService : IMoveCustomerService
{
    public virtual void MoveCustomer(int customerId, Address newAddress)
    {
        // Real operation
    }
}

// The decorator
public class MeasuringMoveCustomerDecorator : IMoveCustomerService
```

```
{
    private readonly IMoveCustomerService decorated;
    private readonly ILogger logger;

    public MeasuringMoveCustomerDecorator(
        IMoveCustomerService decorated, ILogger logger)
    {
        this.decorated = decorated;
        this.logger = logger;
    }

    public void MoveCustomer(int customerId, Address newAddress)
    {
        var watch = Stopwatch.StartNew();

        this.decorated.MoveCustomer(customerId, newAddress);

        this.logger.Log("MoveCustomer executed in " +
            watch.ElapsedMiliseconds + " ms.");
    }
}
```

By wrapping the decorator around the real instance, you can now add this measuring behavior to the class, without any other part of the system to change:

```
IMoveCustomerService service =
    new MeasuringMoveCustomerDecorator(
        new MoveCustomerService(),
        new DatabaseLogger());
```

The previous example did, however, just solve part of the problem (only the SRP part). When writing the code as shown above, you will have to define separate decorators for all operations in the system, and you'll end up with decorators like `MeasuringShipOrderDecorator`, `MeasuringCancelOrderDecorator`, and `MeasuringCancelShippingDecorator`. This lead again to a lot of duplicate code (a violation of the OCP principle), and still needing to write code for every operation in the system. What's missing here is a common abstraction over use cases in the system.

What's missing is an `ICommandHandler<TCommand>` interface.

Let's define this interface:

```
public interface ICommandHandler<TCommand>
{
    void Execute(TCommand command);
}
```

And let's store the method arguments of the `MoveCustomer` method into its own (Parameter Object) class called `MoveCustomerCommand`:

```
public class MoveCustomerCommand
{
    public int CustomerId { get; set; }
    public Address NewAddress { get; set; }
}
```

> ***TIP:*** *This `MoveCustomerCommand` object becomes a message. That's why some postfix this type with 'Message', calling it `MoveCustomerMessage`. Others tend to call it `MoveCustomerRequest`, while others completely remove the post fix and simply call this parameter object `MoveCustomer`. When I wrote this answer initially, I used to use the 'Command' postfix, but nowadays, I tend to go with simply `MoveCustomer`. But whatever you choose, the power here lies in the separation between data (the command/message) and the behavior (the handler), as we'll see next.*

And let's put the behavior of the `MoveCustomer` method in a new class that implements `ICommandHandler<MoveCustomerCommand>`:

```
public class MoveCustomerCommandHandler : ICommandHandler<MoveCustomerCommand>
{
    public void Execute(MoveCustomerCommand command)
    {
        int customerId = command.CustomerId;
        Address newAddress = command.NewAddress;
        // Real operation
    }
}
```

This might look weird at first, but because you now have a general abstraction for use cases, you can rewrite your decorator to the following:

```
public class MeasuringCommandHandlerDecorator<TCommand>
    : ICommandHandler<TCommand>
{
    private ILogger logger;
    private ICommandHandler<TCommand> decorated;

    public MeasuringCommandHandlerDecorator(
        ILogger logger,
        ICommandHandler<TCommand> decorated)
    {
        this.decorated = decorated;
        this.logger = logger;
    }

    public void Execute(TCommand command)
    {
        var watch = Stopwatch.StartNew();

        this.decorated.Execute(command);
```

```
        this.logger.Log(typeof(TCommand).Name + " executed in " +
            watch.ElapsedMiliseconds + " ms.");
    }
}
```

This new `MeasuringCommandHandlerDecorator<T>` looks much like the `MeasuringMoveCustomerDecorator`, but this class can be reused for **all** command handlers in the system:

```
ICommandHandler<MoveCustomerCommand> handler1 =
    new MeasuringCommandHandlerDecorator<MoveCustomerCommand>(
        new MoveCustomerCommandHandler(),
        new DatabaseLogger());

ICommandHandler<ShipOrderCommand> handler2 =
    new MeasuringCommandHandlerDecorator<ShipOrderCommand>(
        new ShipOrderCommandHandler(),
        new DatabaseLogger());
```

This way it will be much, much easier to add cross-cutting concerns to your system. It's quite easy to create a convenient method in your Composition Root that can wrap any created command handler with the applicable command handlers in the system. For instance:

```
private static ICommandHandler<T> Decorate<T>(ICommandHandler<T> decoratee)
{
    return
        new MeasuringCommandHandlerDecorator<T>(
            new DatabaseLogger(),
            new ValidationCommandHandlerDecorator<T>(
                new ValidationProvider(),
                new AuthorizationCommandHandlerDecorator<T>(
                    new AuthorizationChecker(
                        new AspNetUserProvider()),
                    new TransactionCommandHandlerDecorator<T>(
                        decoratee))));
}
```

This method can be used as follows:

```
ICommandHandler<MoveCustomerCommand> handler1 =
    Decorate(new MoveCustomerCommandHandler());

ICommandHandler<ShipOrderCommand> handler2 =
    Decorate(new ShipOrderCommandHandler());
```

If your application starts to grow, however, it can get useful to bootstrap this with a DI Container, because a DI Container can support Auto-Registration. This prevents you from having to make changes to your Composition Root for every new command/handler pair you add to the system.

Most modern, mature DI Containers for .NET have fairly decent support for decorators, and especially Autofac (example) and Simple Injector (example) make it easy to register open-generic decorators.

Unity and Castle, on the other hand, have Dynamic Interception facilities (as Autofac does to btw). Dynamic Interception has a lot in common with decoration, but it uses dynamic-proxy generation under the covers. This can be more flexible than working with generic decorators, but you pay the price when it comes to maintainability, because you often loose type safety and interceptors always force you to take a dependency on the interception library, while decorators are type-safe and can be written without taking a dependency on an external library.

I've been using these types of designs for over a decade now and can't think of designing my applications without it. I've written extensively about these designs, and more recently, I coauthored a book called Dependency Injection Principles, Practices, and Patterns, which goes into much more detail on this SOLID programming style and the design described above (see chapter 10).

Share

Improve this answer

Follow

edited Nov 4, 2022 at 8:10

answered Mar 28, 2012 at 20:19

Steven

172k ● 25 ● 346 ● 447

8 Great Answer Steven. I agree with everything you said here, I also don't agree with catching exceptions just to log and rethrow as I feel there should be one central point in the app domain for doing this, however I do feel there are times when you want to catch a specific exception such as SQLException to retry the action. – OutOFTouch Jan 5, 2013 at 17:48

1 SimpleInjector looks great and is at the top of my list to use, keep up the good work. Here is an example of a decorator with Windsor mikehadlow.blogspot.com/2010/01/… for anyone that is interested. – OutOFTouch Jan 5, 2013 at 18:09

3 @OutOFTouch: Doing a retry of an operation is a very good fit for AOP, but you don't want to wrap such thing about every db operation, but at the transaction boundaries. As a matter of fact, this blog post of mine shows this (take a look at the `DeadlockRetryCommandHandlerDecorator`). – Steven Jan 5, 2013 at 18:21

40 If there is any way to publicize this more, do it. I've been coding for 20 years in a million languages (including Haskell, Racket, Forth) that all claim to change the way you think, and thought I was pretty good at it. This is the first thing in two decades (since 4 horsemen book) that actually changed the way I think, and made me regret every app I've ever written. This should be required reading. – Dax Fohl Mar 31, 2016 at 18:24

Just an observation -- the example for decorators with Simple Injector is now at simpleinjector.readthedocs.io/en/latest/aop.html#decoration – Marcel Popescu May 13, 2019 at 22:16