Lab 01 - Adding Interfaces for Better Control

Objectives

This lab shows how to use interfaces to gain better control over an application. This includes insulating the developer from specific hardware, mitigating risk when using a third-party service, and enhancing performance.

Application Overview

Please refer to the "Lab00-ApplicationOverview.md" file for an overview of the application, the projects, and how to run the application.

As a reminder, there is a folder for .NET 8 (dotnet8). The "Starter" folder contains the code files for this lab.

> Note: The content of the "Starter" folder for Lab 01 is the same as the "Completed" code from Lab 00.

Objectives

1. Remove the need for hardware to be connected to the "COM5" serial port in order to run or test the application. (Note: this will also make it easier to support other hardware.)
2. Use the serial-port hardware as the default system. Be able to override this value for bypassing the hardware or for testing.
3. Make it easier to change from using a specific third-party service for sunrise/sunset data.
4. Sunrise/sunset data does not change for a date and location. Add caching, so you only need to get this data one time per day.

Hints

Hardware Changes

- The COM5 connection is in the `SerialCommander` class.
- Add an abstraction (such as an interface) that has the commander functionality.
- Create a dummy commander that does nothing (or outputs the the console just to see it working).
- Use some dependency injection to inject the commander into the `HouseController` class.

Injecting a Commander

- Use the Property Injection pattern to get the commander into the `HouseController` class.
- The COM port is not opened when the SerialCommander is instantiated, so it can be safely "new"ed up as long as the commander is overridden before it is used.

Changing the Sunset Provider

- The `SolarServiceSunsetProvider` class makes calls to the third-party service.

- Add an abstraction (such as an interface) that has the sunset provider functionality.
- Use some dependency injection to inject the sunset provider into the `Schedule` class.
- Use the `SolarCalculator` NuGet package to create a class that calculates sunrise and sunset times locally.

Caching Sunset Data

- The Decorator Pattern can be really useful for adding functionality to an existing class (or interface).
- The sunrise/sunset data only changes when the location and/or date changes.

If this is enough information for you, and you want a bit of a challenge, **stop reading here**. Otherwise, continue reading for step-by-step instructions.

Step-by-Step

Keep scrolling for the step-by-step walkthrough. Or use the links below to jump to a section:

Step-by-Step - Hardware Changes

Start the SolarCaculator.Service and HouseControlAgent projects (as described in Lab 00 - Application Overview). The application fails with the following exception:

```
System.IO.FileNotFoundException: 'Could not find file 'COM5'.'
```

If you run HouseControlAgent in a debugger, it will take you to the `SerialCommander.cs` file in the `HouseControl.Library` project: [SolutionPath]\HouseControl.Library\Commanders\Serial\SerialCommander.cs.

The `SerialCommander` class has one method: `SendCommand`. Extract this into an interface to make it easier to create a fake commander (and to swap out to a different commander later).

1. Create a new file in the HouseControl.Library project's "Commanders" folder called `ICommander`.

2. Create an interface that contains the `SendCommand` method. Here is the completed code code:

```
namespace HouseControl.Library;

public interface ICommander
{
    Task SendCommand(int deviceNumber, DeviceCommand command);
}
```

Note: The namespace here is `HouseControl.Library` and does not follow the folder structure. I prefer to have a minimal number of namespaces unless complexity requires it. But feel free to use the namespace naming method of your choice.

> As an alternative to creating the interface manually, both Visual Studio 2022 and Visual Studio Code offer shortcuts to extract an interface. These can be extremely helpful, particularly for more complex interfaces. If you extract an interface from the SerialCommander class, you will want to pay attention to the interface name, where it puts the file, and the namespace. You may need to move the file after it is created and adjust namespaces.

3. Specify that the `SerialCommander` class implements the new `ICommander` interface by adding it to the class declaration.

```
public class SerialCommander : ICommander
{
    // code omitted
}
```

Since `SerialCommander` already has the `SendCommand` method, no other changes need to be made.

> If you used your IDE to extract an interface, this step is done for you automatically.

Now that you have an interface, update the `HouseController` class to use it.

4. Open the `HouseController.cs` file (in the same `HouseController.Library` project).
5. Change the type of the private `commander` field at the top of the class from `SerialCommander` to `ICommander`.

```
public class HouseController
{
    private readonly ICommander commander;
    // other code omitted
}
```

If you build and run the code at this point, you will get the same "FileNotFoundException". That's good, you haven't changed the functionality (yet).

6. Look at the constructor for the `HouseController` class.

```
public HouseController(Schedule schedule)
{
    commander = new SerialCommander();
```

```
    // other code omitted
}
```

Rather that newing up a `SerialCommander` in the contructor, inject the commander through a parameter.

7. Add an `ICommander` parameter to the constructor and set the local field.

```
public HouseController(Schedule schedule, ICommander commander)
{
    this.commander = commander;
    // other code omitted
}
```

> `this.commander` refers to the class-level field.
> `commander` (without "this") refers to the constructor parameter.

8. Build the application. At this point, you get a build failure because you changed the constructor signature.

9. Open the `Program.cs` file in the `HouseControlAgent` project, and look at the `InitializeHouseController` method.

```
private static HouseController InitializeHouseController()
{
    //45.6382,-122.7013 = Vancouver, WA, USA
    //28.4810,-81.5074 = Orlando, FL
    //28.4672,-81.4687 = Royal Pacific Hotel

    var fileName = AppDomain.CurrentDomain.BaseDirectory + "ScheduleData";
    var sunsetProvider = new SolarServiceSunsetProvider(51.520,-0.0963);
    var schedule = new Schedule(fileName, sunsetProvider);
    var controller = new HouseController(schedule);

    var sunset = sunsetProvider.GetSunset(DateTime.Today.AddDays(1));
    Console.WriteLine($"Sunset Tomorrow: {sunset:G}");

    return controller;
}
```

10. Create a new instance of the `SerialCommander` and pass it to the `HouseController` constructor.

```
var commander = new SerialCommander();
var controller = new HouseController(schedule, commander);
```

11. Build and run the application.

You get a successful build at this point but still have the runtime exception. This is okay, because you have not changed any functionality yet, just rearranged the pieces a bit.

12. Create a `FakeCommander` class.
    In the `HouseController.Library` project, create a new file in the `Commanders` folder called `FakeCommander`.

```
namespace HouseControl.Library;

public class FakeCommander
{
}
```

13. Specify that the `FakeCommander` class implements the `ICommander` interface.

```
public class FakeCommander : ICommander
{
}
```

14. Implement the interface.

> In Visual Studio 2022, I like to use the shorcut for this. Put the cursor on `ICommander`, then press "Ctrl+." to bring up the lightbulb help. Then choose "Implement interface". This will create placeholders for any interface members that do not already exist.

```
public class FakeCommander : ICommander
{
    public Task SendCommand(int deviceNumber, DeviceCommand command)
    {
        throw new NotImplementedException();
    }
}
```

15. Update the `SendCommand` method to output the device and command to the console.

```
public Task SendCommand(int deviceNumber, DeviceCommand command)
{
    Console.WriteLine($"Device {deviceNumber}: {command}");
    return Task.CompletedTask;
}
```

This will let us know that the `SendCommand` method is actually getting called.

> Note that this returns a completed task from this method. Since this is an asynchronous method, it needs to return a Task. Because there is no async code in this method, it can return `Task.CompletedTask`. This gives us a Task that is already in the "completed successfully" state. This satisfies the return value with minimal overhead.

16. Use the `FakeCommander` class in the `HouseControlAgent` application.

In the `Program.cs` file of the `HouseControlAgent` project, change the `SerialCommander` to a `FakeCommander`.

```
var commander = new FakeCommander();
var controller = new HouseController(schedule, commander);
```

17. Build and run the application.

```
Initializing Controller
Sunset Tomorrow: 11/19/2024 6:18:16 PM
Device 5: On
11/18/2024 12:21:36 PM - Device: 5, Command: On
Device 5: Off
11/18/2024 12:21:36 PM - Device: 5, Command: Off
Initialization Complete
```

The messages "Device 5: On" and "Device 5: Off" come from the `FakeCommander` class. This lets us know that the `SendCommand` method is being called.

If you let the `HouseControlAgent` continue to run, you will see the various test items processing every minute for about 5 minutes.

```
Initializing Controller
Sunset Tomorrow: 11/19/2024 6:18:16 PM
Device 5: On
11/18/2024 12:21:36 PM - Device: 5, Command: On
Device 5: Off
11/18/2024 12:21:36 PM - Device: 5, Command: Off
Initialization Complete
Device 3: On
11/18/2024 12:22:36 PM - Device: 3, Command: On
Schedule Items Processed: 1 - Total Items: 20 - Active Items: 6
Device 5: On
11/18/2024 12:23:36 PM - Device: 5, Command: On
Schedule Items Processed: 1 - Total Items: 19 - Active Items: 5
```

```
Device 3: Off
11/18/2024 12:24:36 PM - Device: 3, Command: Off
Schedule Items Processed: 1 - Total Items: 18 - Active Items: 4
Device 5: Off
11/18/2024 12:25:36 PM - Device: 5, Command: Off
Schedule Items Processed: 1 - Total Items: 17 - Active Items: 3
Schedule Items Processed: 0 - Total Items: 16 - Active Items: 2
```

So you have successfully removed the need to have a physical device connected in order to run the system and test the overall functionality.

Step-by-Step - Injecting a Commander

You are currently passing the commander to the `HouseController` as a constructor parameter (i.e., using the Constructor Injection pattern). This works well in scenarios where you want to force someone to choose a dependency.

But in this situation, you have a commander that you always want to use at runtime (the `SerialCommander`), but you want to be able to override it when you are testing or do not have the hardware available (with a `FakeCommander`). For this, you can use a different pattern: Property Injection.

The overall idea is create a property that is initialized with a default dependency. If you do nothing, then the default is used. But you can change the property before using the class to inject different behavior.

1. Open the `HouseController.cs` file in the `HouseControl.Library` project.

2. Look at the `commander` field and the constructor.

```
public class HouseController
{
    private readonly ICommander commander;
    // other code omitted

    public HouseController(Schedule schedule, ICommander commander)
    {
        this.commander = commander;

        // other code omitted
   }
   // other code omitted
}
```

3. Remove the `commander` parameter from the constructor.

```
public class HouseController
{
    private readonly ICommander commander;
    // other code omitted

    public HouseController(Schedule schedule)
    {
        //this.commander = commander;

        // other code omitted
   }
   // other code omitted
}
```

3. Add a public `Commander` property and use the existing field as a backing field.

```
private ICommander? commander;
public ICommander Commander
{
    get => commander;
    set => commander = value;
}
```

Notice that the backing field no longer readonly, and it is also now nullable (as denoted by the "?"). In Visual Studio 2022, there will also be a warning on the getter that `commander` may be null.

4. If `commander` is null in the getter, set the value to a new `SerialCommander`.

```
private ICommander? commander;
public ICommander Commander
{
    get
    {
         if (commander is null)
            commander = new SerialCommander();
        return commander;
    }
    set => commander = value;
}
```

5. Simplify the null check by using the null conditional (`??`) operator.

```
private ICommander? commander;
public ICommander Commander
{
    get
    {
        commander = commander ?? new SerialCommander();
        return commander;
    }
    set => commander = value;
}
```

6. This can be further simplified by combining the assignment (`=`) with the null conditional (`??`).

```
private ICommander? commander;
public ICommander Commander
{
    get
    {
        commander ??= new SerialCommander();
        return commander;
    }
    set => commander = value;
}
```

7. This can be further simplified by combining the assignment line with the return line.

```
private ICommander? commander;
public ICommander Commander
{
    get
    {
        return commander ??= new SerialCommander();
    }
    set => commander = value;
}
```

8. Since you are back to a single-line getter, you can change this back to an expression-bodied member.

```
private ICommander? commander;
public ICommander Commander
{
    get => commander ??= new SerialCommander();
```

```
        set => commander = value;
    }
```

This code has the same effect. The first time we use the `Commander` property in the code, it will check the backing field, if the field is not null, then it will use that value. If the field is null, then it will new up a `SerialCommander`, assign it to the backing field, and return that object.

9. The last step is to change the references to the field (`commander`) to reference the property (`Commander`). This appears only in one method: `SendCommand`.

```
public async Task SendCommand(int device, DeviceCommand command)
{
    //await commander.SendCommand(device, command);
    await Commander.SendCommand(device, command);
    Console.WriteLine($"{DateTime.Now:G} - Device: {device}, Command: {command}");
}
```

If you build at this point, you will get a build failure (since you changed the constructor signature).

10. Open the `Program.cs` file in the `HouseControlAgent` project.

11. In the `InitializeHouseController` method, remove the `commander` parameter from the `HouseController` constructor call.

```
//var commander = new FakeCommander();
//var controller = new HouseController(schedule, commander);
var controller = new HouseController(schedule);
```

12. Run the application.

You get the same exception that you've seen before.

```
System.IO.FileNotFoundException: 'Could not find file 'COM5'.'
```

This tells you that you are using the `SerialCommander` (the default).

13. After creating the `HouseController`, set the `Commander` property to a `FakeCommander`.

```
var controller = new HouseController(schedule);
controller.Commander = new FakeCommander();
```

14. Run the application.

```
Initializing Controller
Sunset Tomorrow: 11/19/2024 6:18:16 PM
Device 5: On
11/18/2024 12:31:22 PM - Device: 5, Command: On
Device 5: Off
11/18/2024 12:31:22 PM - Device: 5, Command: Off
Initialization Complete
```

Now we are using the `FakeCommander` once again.

Property Injection gives you a good way to set up a default value that will be used if you do nothing. But it still gives you a place to override the dependency behavior if you need to.

Step-by-Step - Changing the Sunset Provider

Another reason to consider adding an interface and/or using dependency injection is when you have a dependency that you do not fully control. In this code, the `SolarCalulator.Service` represents a third-party service (even though you have a replicated local copy here).

> When I first wrote this application, I added an interface between my application and the service in order to mitigate risks. And I am glad that I did. One day I was updating the application, and the service started returning "500" errors. Since I had the interface in place, I could quickly swap in a fake sunset provider so I could continue working on the feature I wanted to change. Then I went back and added a local sunrise/sunset calculation based on a NuGet package. You will replicate this here.

1. Open the `SolarServiceSunsetProvider.cs` file in the `HouseControl.Sunset` project.

This class has a number of methods, but the ones that we care about are `GetSunrise` and `GetSunset`.

```
public DateTimeOffset GetSunrise(DateTime date)
{
    string serviceData = GetServiceData(date, latitude, longitude);
    string sunriseTimeString = ParseSunriseTime(serviceData);
    DateTime sunriseTime = ToLocalTime(sunriseTimeString, date);
    return new DateTimeOffset(sunriseTime);
}

public DateTimeOffset GetSunset(DateTime date)
{
    string serviceData = GetServiceData(date, latitude, longitude);
    string sunsetTimeString = ParseSunsetTime(serviceData);
    DateTime sunsetTime = ToLocalTime(sunsetTimeString, date);
    return new DateTimeOffset(sunsetTime);
}
```

2. Create an interface called `ISunsetProvider` that has these methods. This will also be in the `HouseControl.Sunset` project.

```
namespace HouseControl.Sunset;

public interface ISunsetProvider
{
    DateTimeOffset GetSunrise(DateTime date);
    DateTimeOffset GetSunset(DateTime date);
}
```

> `DateTimeOffset` is a `DateTime` that also includes timezone information.

3. Update the `SolarServiceSunsetProvider` class to indicate that it implements the `ISunsetProvider` interface.

```
public class SolarServiceSunsetProvider : ISunsetProvider
```

4. Open the NuGet manager (or look in the project file) to see the packages for the `HouseControl.Sunset` project.

You will find a reference to `SolarCalculator` by Daniel M. Porrey. This is the package you will use to calculate sunrise and sunset.

The main type in this package is the `SolarTimes` type, so use this for naming.

5. Create a `SolarTimesSunsetProvider` class in the same project.

```
namespace HouseControl.Sunset;

public class SolarTimesSunsetProvider
{
}
```

6. Add a reference to the `ISunsetProvider` interface.

```
public class SolarTimesSunsetProvider : ISunsetProvider
{
}
```

7. Implement the 2 methods of the interface.

```
public class SolarTimesSunsetProvider : ISunsetProvider
{
    public DateTimeOffset GetSunrise(DateTime date)
    {
        throw new NotImplementedException();
    }

    public DateTimeOffset GetSunset(DateTime date)
    {
        throw new NotImplementedException();
    }
}
```

The `SolarTimes` constructor takes a date, latitude, and longitude values.

8. Add fields for latitude and longitude values (type double), and a constructor that sets those values.

Note: the `SolarServiceSunsetProvider` has these same fields and constructor parameters, so you can copy the bulk of it from that class.

```
public class SolarTimesSunsetProvider : ISunsetProvider
{
    private readonly double latitude;
    private readonly double longitude;

    public SolarTimesSunsetProvider(double latitude, double longitude)
    {
        this.latitude = latitude;
        this.longitude = longitude;
    }
    // other code omitted
}
```

9. Add a using statement for `Innovative.SolarCalculator` for easy access to the NuGet package.

```
using Innovative.SolarCalculator;
```

10. New up a `SolarTimes` object, and get return the `Sunrise` and `Sunset` properties.

```
public DateTimeOffset GetSunrise(DateTime date)
{
```

```
    var solarTimes = new SolarTimes(date, latitude, longitude);
    return new DateTimeOffset(solarTimes.Sunrise);
}

public DateTimeOffset GetSunset(DateTime date)
{
    var solarTimes = new SolarTimes(date, latitude, longitude);
    return new DateTimeOffset(solarTimes.Sunset);
}
```

> `DateTimeOffset` is a `DateTime` that also includes timezone information. If no specific timezone is provided, then it uses the timezone associated with the machine where the code is running.

11. The sunset provider is already injected into the `Schedule` and `ScheduleHelper` classes in the `HouseControl.Library` project. Update these classes to use the `ISunsetProvider` interface rather than the `SolarSunsetServiceProvider`. (Steps below.)

12. Open the `ScheduleHelper` class in the `HouseControl.Library` project.

13. Change the `SunsetProvider` property to the `ISunsetProvider` interface, and also update the constructor parameter.

```
public class ScheduleHelper
{
    //private readonly SolarServiceSunsetProvider SunsetProvider;
    private readonly ISunsetProvider SunsetProvider;

    public ScheduleHelper(ISunsetProvider sunsetProvider)
    {
        this.SunsetProvider = sunsetProvider;
    }
    // other code omitted
}
```

14. Open the `Schedule` class in the `HouseControl.Library` project.

15. Update the constructor to use the `ISunsetProvider` interface for the parameter.

```
public Schedule(string filename, ISunsetProvider sunsetProvider)
{
    this.scheduleHelper = new ScheduleHelper(sunsetProvider);
    this.filename = filename;
    LoadSchedule();
}
```

16. Build and run the application.

```
Initializing Controller
Sunset Tomorrow: 11/19/2024 6:18:16 PM
Device 5: On
11/18/2024 12:39:39 PM - Device: 5, Command: On
Device 5: Off
11/18/2024 12:39:39 PM - Device: 5, Command: Off
Initialization Complete
```

At this point, you have not changed the functionality. The code still uses the service for sunrise and sunset data.

17. Stop the application, and stop the service.

In the terminal window where the service is running, press "Ctrl+c" to stop the service (or close the terminal window).

18. Re-run the application.

You should get the following exception:

```
System.AggregateException: 'One or more errors occurred. (No connection could be
made because the target machine actively refused it. (localhost:8973))'
```

This lets you know that the operation failed because the service is not running.

19. Stop the application.

20. Open the `Program.cs` file in the `HouseControlAgent` project.

21. Change the `sunsetProvider` variable from a `SolarServiceSunsetProvider` to `SolarTimesSunsetProvider`.

```
//var sunsetProvider = new SolarServiceSunsetProvider(28.4810,-81.5074);
var sunsetProvider = new SolarTimesSunsetProvider(28.4810,-81.5074);
var schedule = new Schedule(fileName, sunsetProvider);
var controller = new HouseController(schedule);
controller.Commander = new FakeCommander();
```

22. Re-run the application.

```
Initializing Controller
Sunset Tomorrow: 11/19/2024 6:18:16 PM
```

```
Device 5: On
11/18/2024 12:41:26 PM - Device: 5, Command: On
Device 5: Off
11/18/2024 12:41:26 PM - Device: 5, Command: Off
Initialization Complete
```

This has the locally-calculated sunset time. The application no longer relies on a third-party service.

The addition of the interface gives you options for calculating sunrise and sunset times. In addition, you can create a fake sunset provider for testing of the `Schedule` and `ScheduleHelper` classes. (Testing these classes is part of Lab 02).

Step-by-Step - Caching Sunset Data

As a last step in this lab, create a cache for the sunrise and sunset times. These times stay the same throughout the day, so there is no need to refetch or recalculate the values every time you need them. Instead, use a simple cache based on the current date.

For this, you can use the Decorator Pattern. This is a pattern where you wrap an existing object and provide new functionality. This is sometimes called an "interceptor" because you intercept the call to the real object and alter the functionality.

The basics steps here are to create a caching provider that implements the `ISunsetProvider` interface. This class will also take an `ISunsetProvider` (the "real" provider) as a constructor parameter.

To use the cache, you change the composition root (where you put the loosely-coupled pieces together). This means that you do not need to change any of the classes that implement `ISunsetProvider` or any of the classes that depend on an `ISunsetProvider`.

1. Create a new `CachingSunsetProvider` class in the `HouseControl.Sunset` project.

```
namespace HouseControl.Sunset;

public class CachingSunsetProvider
{
}
```

2. Implement the `ISunsetProvider` interface.

```
public class CachingSunsetProvider : ISunsetProvider
{
    public DateTimeOffset GetSunrise(DateTime date)
    {
        throw new NotImplementedException();
    }
```

```
    public DateTimeOffset GetSunset(DateTime date)
    {
        throw new NotImplementedException();
    }
}
```

3. Create a field to hold the "real" sunset provider, and set it with a constructor parameter.

```
public class CachingSunsetProvider : ISunsetProvider
{
    private readonly ISunsetProvider wrappedProvider;

    public CachingSunsetProvider(ISunsetProvider sunsetProvider)
    {
        wrappedProvider = sunsetProvider;
    }
    // other code omitted.
}
```

4. Add fields for sunrise, sunset, and date to hold the cached values.

```
    private DateTime dataDate;
    private DateTimeOffset sunrise;
    private DateTimeOffset sunset;
```

5. Add a method called `ValidateCache` that will make sure the cache is current.

If the `dataDate` matches the requested date, then the data is current. If the `dataDate` does **not** match the requested date, then update the values using the wrapped sunset provider.

```
private void ValidateCache(DateTime date)
{
    if (dataDate != date)
    {
        sunrise = wrappedProvider.GetSunrise(date);
        sunset = wrappedProvider.GetSunset(date);
        dataDate = date;
    }
}
```

6. Update the `GetSunrise` and `GetSunset` methods to validate the cache and return the value in the cache.

```
public DateTimeOffset GetSunrise(DateTime date)
{
    ValidateCache(date);
    return sunrise;
}

public DateTimeOffset GetSunset(DateTime date)
{
    ValidateCache(date);
    return sunset;
}
```

This is a fairly naive implementation of a cache. It only works with one date at a time, and it also assumes that the location stays constant. In a more complex scenario, the cache could be extended to accommodate additional parameters.

The last step is to use the cache.

7. Open the `Program.cs` file in the `HouseControlAgent` project.
8. In the `InitializeHouseController` method, create a new variable for the caching sunset provider, and use the `SolarTimesSunsetProvider` as a constructor parameter.

```
    var sunsetProvider = new SolarTimesSunsetProvider(51.520,-0.0963);
    var cachingSunsetProvider = new CachingSunsetProvider(sunsetProvider);
```

9. Pass the caching sunset provider as a parameter to the schedule.

```
    var sunsetProvider = new SolarTimesSunsetProvider(51.520,-0.0963);
    var cachingSunsetProvider = new CachingSunsetProvider(sunsetProvider);
    //var schedule = new Schedule(fileName, sunsetProvider);
    var schedule = new Schedule(fileName, cachingSunsetProvider);
```

10. Update the `GetSunset` method call to use the caching sunset provider.

```
var sunset = cachingSunsetProvider.GetSunset(DateTime.Today.AddDays(1));
Console.WriteLine($"Sunset Tomorrow: {sunset:G}");
```

11. Build and run the application.

```
```

```
Initializing Controller
Sunset Tomorrow: 11/19/2024 6:18:16 PM
Device 5: On
11/18/2024 12:47:00 PM - Device: 5, Command: On
Device 5: Off
11/18/2024 12:47:00 PM - Device: 5, Command: Off
Initialization Complete
```

There is no apparent change to the output.

12. Show the cache working.

Here are a few things that will to show that the cache is working.

- In the `InitializeHouseController` method, double up the call to `GetSunset`.

```
var sunset = cachingSunsetProvider.GetSunset(DateTime.Today.AddDays(1));
Console.WriteLine($"Sunset Tomorrow: {sunset:G}");
sunset = cachingSunsetProvider.GetSunset(DateTime.Today.AddDays(1));
Console.WriteLine($"Sunset Tomorrow: {sunset:G}");
```

- In the `SolarTimesSunsetProvider` class, update the `GetSunset` method to output to the console.

```
public DateTimeOffset GetSunset(DateTime date)
{
    Console.WriteLine("Calculating Sunset");
    var solarTimes = new SolarTimes(date, latitude, longitude);
    return new DateTimeOffset(solarTimes.Sunset);
}
```

- Re-run the application.

```
Initializing Controller
Calculating Sunset
Sunset Tomorrow: 11/19/2024 6:18:16 PM
Sunset Tomorrow: 11/19/2024 6:18:16 PM
Device 5: On
11/18/2024 12:48:11 PM - Device: 5, Command: On
Device 5: Off
11/18/2024 12:48:11 PM - Device: 5, Command: Off
Initialization Complete
```

Even though we call `GetSunset` twice, "Calculating Sunset" only appears once in our output. The second call uses the cache value from the caching sunset provider.

- Before moving on, remove the code that you added in step 12 (the duplicate `GetSunset` call and console output).

By using the Decorator Pattern in conjuction with interfaces and dependency injection, you were able to add caching to an existing class. Note that you did not need to change the existing sunset providers (`SolarTimeSunsetProvider` or `SolarServiceSunsetProvider`). You also did not need to change the classes that use the sunset provider (`Schedule` and `ScheduleHelper`). In addition, the caching sunset provider will work with any existing or new provider that you may come up with.

Pretty neat, huh?

Wrap Up

In this lab you saw how interfaces and dependency injection can make applications easier to change.

You removed the need to have hardware connected on COM5 for the application to work. This lets you run the application when hardware is not available and also helps with unit testing (which you'll see in a later lab).

Since you want to use the hardware in production, you made it a default by using Property Injection. If you do nothing, then the serial hardware is used. But you still have a "seam" in the code where you can inject a fake object when needed.

You removed a dependency on a third-party service. It's best to have a bit of insulation around things that you do not control. That way if the functionality changes (or stops working entirely), you can easily swap in functionality from a different source -- or at least use a fake source until you can find a new one.

Finally, you used the Decorator Pattern to add caching to the sunset provider. This functionality will work with any sunset provider, and you did not need to change any of the existing objects in order to get this to work. The combination of interfaces and dependency injection that you already had in place for other purposes made this very easy.

In the next lab, you will write unit tests and find that there are benefits to these new interfaces in testing as well.

END - Lab 01 - Adding Interfaces for Better Control