Lab 02 - Property Injection and Handling DateTime.Now

Objectives

In this lab, you will use the Property Injection pattern to make it easy to set up default behavior. In addition, you will see the difficulty of trying to unit test `DateTime.Now`. Also using the Property Injection pattern, you can set things up so that the current date/time is used by default, but you can also change out that value for unit testing purposes. These techniques are transferable to other types of static dependencies as well.

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, the "Starter" folder contains the code files for this lab.

> The state of the "Starter" files for Lab 02 are the same as the "Completed" files for Lab 01. So if you completed Lab 01, you can continue coding in the same solution.

Objectives

1. Change the current application to use the serial-port hardware by default. Also allow this to be overridden to bypass the hardware for testing and development purposes.
2. Create an abstraction for `DateTimeOffset.Now` using the `TimeProvider` type that is part of .NET 8 and .NET 9.
3. Create tests to test for `Now` and `IsInFuture` methods in the `ScheduleHelper` class.

Hints

Injecting a Commander

- Use the Property Injection pattern to set a good default for the commander in 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.
- Update the composition root to set the new property to a fake object when necessary.

Injecting DateTime.Now

- Find all references to `DateTimeOffset.Now` in the code. These should be limited to 2 classes: `Schedule` and `ScheduleHelper`.
- Add a field for the abstract `TimeProvider` type (a type that was added in .NET 8).
- `TimeProvider` has a static `System` property that uses the real time.

Unit Testing DateTime.Now

- A unit testing project has been stubbed out using xUnit: `HouseControl.Library.Test`. If you are more comfortable with a different framework (such as NUnit or MSTest), feel free to use that instead.
- `DateTimeOffset` is a `DateTime` that also includes timezone information.
- Use the `Microsoft.Extensions.TimeProvider.Testing` NuGet package. You can easily find this by searching for "FakeTimeProvider" in the NuGet Package Manager.

Other Hints

- The application only deals with the local timezone (meaning, the timezone of the machine the application runs on). The application does not need to support multiple locations or alternate time zones.
- When setting times for tests, you will need to include a time zone offset (which is a time span). You may want to create a field such as the following:

```
TimeSpan timeZoneOffset = new TimeSpan(-2, 0, 0);
```

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 - 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
}
```

4. 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.

5. 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;
}
```

6. 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;
}
```

7. 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;
}
```

8. 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;
}
```

9. 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.

10. 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).

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

12. 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);
```

13. 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).

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

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

15. Run the application.

```
Initializing Controller
Sunset Tomorrow: 11/1/2025 6:14:04 PM
Device 5: On
11/2/2025 10:44:54 AM - Device: 5, Command: On
Device 5: Off
11/2/2025 10:44:54 AM - 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 - Injecting DateTime.Now

1. Centralize all calls to `DateTimeOffset.Now`. This will make the abstraction easier.
2. Open the `ScheduleHelper.cs` file in the `HouseControl.Library` project.
3. Create a helper method on the `ScheduleHelper` class for `Now`.

```
public DateTimeOffset Now()
{
    return DateTimeOffset.Now;
}
```

4. In the `ScheduleHelper` class, search for references to `DateTimeOffset.Now` and update them to use `Now()`.

```
public DateTimeOffset Tomorrow()
{
    //return new DateTimeOffset(
    //    DateTimeOffset.Now.Date.AddDays(1),
    //    DateTimeOffset.Now.Offset);
    return new DateTimeOffset(
        Now().Date.AddDays(1),
        Now().Offset);
}
```

```
public bool IsInFuture(DateTimeOffset checkTime)
{
    // return checkTime > DateTime.Now;
    return checkTime > Now();
}
```

You should have 3 replacements in `ScheduleHelper`.

5. In the `Schedule` class, search for references to `DateTimeOffset.Now` and update them to use `scheduleHelper.Now`.

```
//DateTimeOffset today = DateTimeOffset.Now.Date;
DateTimeOffset today = scheduleHelper.Now().Date;
```

```
//(si.Info.EventTime - DateTimeOffset.Now).Duration() < TimeSpan.FromSeconds(30))
(si.Info.EventTime - scheduleHelper.Now()).Duration() < TimeSpan.FromSeconds(30))
```

```
//while (currentItem.Info.EventTime < DateTimeOffset.Now)
while (currentItem.Info.EventTime < ScheduleHelper.Now())
```

There are 3 replacements in `Schedule`.

You should be able to buid and run the application at this point.

6. In the `ScheduleHelper` class, create a new property to hold a `TimeProvider`.

```
private TimeProvider? helperTimeProvider;
public TimeProvider HelperTimeProvider
```

```
{
    get { return helperTimeProvider; }
    set { helperTimeProvider = value; }
}
```

7. Update the `get` method so that if the backing field is null, set the backing field to the `CurrentTimeProvider` and return the field.

```
get
{
    if (helperTimeProvider is null)
        helperTimeProvider = TimeProvider.System;
    return helperTimeProvider;
}
```

> The result is that the the System (a.k.a. "real" time provider) is used by default. So if you do not explicitly set the `HelperTimeProvider` property, it uses the current time. This is the behavior you want in production. But for testing, you now have a property to override to provide your own value of `Now`.

8. Using the technique from Lab 01, shorten the code using the null coalescing operator (`??`).

```
get
{
    return helperTimeProvider ??= TimeProvider.System;
}
```

9. The property `get`/`set` can also be updated to use expression-bodied members. Here is the final property:

```
private TimeProvider? helperTimeProvider;
public TimeProvider HelperTimeProvider
{
    get => helperTimeProvider ??= TimeProvider.System;
    set => helperTimeProvider = value;
}
```

10. Update the `Now` method to use the new property.

```
public DateTimeOffset Now()
{
    return HelperTimeProvider.GetLocalNow();
}
```

> Note: The `GetLocalNow` method returns a `DateTimeOffset` with the current time and time zone offset.

With the new property in place for getting the current time, you now have a plug-in point for setting your own "now" time in unit tests.

Step-by-Step - Unit Testing DateTime.Now

1. Open the `ScheduleHelper.Tests.cs` file in the `HouseControl.Library.Tests` project. This file does not have any tests stubbed out.

2. Create a unit test to make sure that `Now` returns the configured (i.e. fake) time.

```
[Fact]
public void Now_ReturnsConfiguredTime()
{
    // Arrange

    // Act

    // Assert

}
```

> Note: I like to use the Arrange/Act/Assert arrangement for unit testing. This gives a place to configure the test (Arrange), call the method/property that I am testing (Act), and checking the results (Assert).

3. In the `Arrange` section, create an instance of the `ScheduleHelper`.

```
    // Arrange
    var helper = new ScheduleHelper();
```

The constructor needs an `ISunsetProvider`. This is a great chance to create a fake object for testing.

4. Based on your previous experience at creating fake objects, create a new `FakeSunsetProvider` class in the testing project. This should implement the `ISunsetProvider` interface.

```
using HouseControl.Sunset;

namespace HouseControl.Library.Tests;

public class FakeSunsetProvider : ISunsetProvider
{
    public DateTimeOffset GetSunrise(DateTime date)
```

```
        {
            throw new NotImplementedException();
        }

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

You will need to a `using` statement to reference the `HouseControl.Sunset` project.

5. Create a private `TimeSpan` object to hold a time zone offset (as mentioned in the Hints section above). Then use that value to create fake sunrise and sunset times that can be used for testing.

> The parameters for the `DateTimeOffset` constructor are as follows: year, month, day, hour, minute, second, timezoneoffset.

Note that these times are not used for the current tests, so the specific sunrise/sunset values are not important.

```
public class FakeSunsetProvider : ISunsetProvider
{
    private TimeSpan timeZoneOffset = new(-2, 0, 0);

    public DateTimeOffset GetSunrise(DateTime date)
    {
        return new DateTimeOffset(2025, 11, 02, 06, 15, 55, timeZoneOffset);
    }

    public DateTimeOffset GetSunset(DateTime date)
    {
        return new DateTimeOffset(2025, 11, 02, 18, 15, 55, timeZoneOffset);
    }
}
```

As an alternate to using a fake object, you could also use a mocking framework, such as Moq, but that is outside the scope of this lab.

6. Use the `FakeSunsetProvider` in the unit test.

```
        // Arrange
        var helper = new ScheduleHelper(new FakeSunsetProvider());
```

7. Next create a `DateTimeOffset` to represent the current date/time (the specific value does not matter). You can also create a `timeZoneOffset` field in this calss to help out.

```
private TimeSpan timeZoneOffset = new(-2, 0, 0);

[Fact]
public void Now_ReturnsConfiguredTime()
{
    // Arrange
    var helper = new ScheduleHelper(new FakeSunsetProvider());
    var now = new DateTimeOffset(2025, 11, 01, 14, 22, 22, timeZoneOffset);

    // Act

    // Assert

}
```

8. In the `Act` section, call the ScheduleHelper's `Now` method.

```
    // Act
    var actual = helper.Now();
```

9. In the `Assert` section, compare the `now` and `actual` values.

```
    // Assert
    Assert.Equal(now, actual);
```

10. Run the test.

At this point, the test most likely will fail. The reason for the failure is that this test relies on the default time provider that is used by default, and that points to the current time. You can create your own `TimeProvider` and override certain methods.

But making our own `TimeProvider` is more complicated than I would like it to be. Fortunately, there is a NuGet package to bring in a fake time provider specifically for unit testing.

11. Add the `Microsoft.Extensions.TimeProvider.Testing` NuGet package. You can easily find this by searching for "FakeTimeProvider" in the NuGet Package Manager.

12. After creating the `ScheduleHelper`, update the `HelperTimeProvider` property to use a fake time provider. The `FakeTimeProvider` comes from the NuGet package, and we can pass in the `DateTimeOffset` value to use as the "current" time.

```
var helper = new ScheduleHelper(new FakeSunsetProvider());
var now = new DateTimeOffset(2025, 11, 01, 14, 22, 22, timeZoneOffset);
helper.HelperTimeProvider = new FakeTimeProvider(now);
```

13. Rerun the test. It should now pass.

We are going to use many of the same variables for testing different scenarios. Rather than duplicating things, let's extract a few items.

14. Create a new test for the `Tomorrow` method. Start with the `Arrange` section from the previous test.

```
[Fact]
public void Tomorrow_ReturnsConfiguredTimePlusOneDay()
{
    // Arrange
    var helper = new ScheduleHelper(new FakeSunsetProvider());
    var now = new DateTimeOffset(2025, 11, 01, 14, 22, 22, timeZoneOffset);
    helper.HelperTimeProvider = new FakeTimeProvider(now);

}
```

15. Also in the `Arrange` section, add a variable for the `expected` value with is "tomorrow's" date with 0s for the time portion.

```
    var expected = new DateTimeOffset(2025, 11, 02, 00, 00, 00, timeZoneOffset);
```

16. In the `Act` section, call the `Tomorrow` method.

```
    // Act
    var actual = helper.Tomorrow();
```

17. In the `Assert` section, compare the expected and actual values.

```
    // Assert
    Assert.Equal(expected, actual);
```

18. One change I like to make for readability is to move the `now` and `expected` values closer to each other so it is easier to compare the values by eyey. Here is the completed test.

```
[Fact]
public void Tomorrow_ReturnsConfiguredTimePlusOneDay()
{
    // Arrange
    var now =      new DateTimeOffset(2025, 11, 01, 14, 22, 22, timeZoneOffset);
    var expected = new DateTimeOffset(2025, 11, 02, 00, 00, 00, timeZoneOffset);
    var helper = new ScheduleHelper(new FakeSunsetProvider());
    helper.HelperTimeProvider = new FakeTimeProvider(now);

    // Act
    var actual = helper.Tomorrow();

    // Assert
    Assert.Equal(expected, actual);
}
```

19. Re-run the tests. Both tests should pass.

20. For the final test, create a test for `IsInFuture` that expects a `true` result.

```
    [Fact]
    public void IsInFuture_WithFutureDate_ReturnsTrue()
    {

    }
```

> I use a 3-part naming scheme to help me organize tests. The first part is the "unit under test" (the IsInFuture method). The second part is the condition (the date is in the future). The third part is the expected result (returns "true"). On main advantage is that when looking at the test explorer, I often do not need to look at the details of the test since the name gives a lot of information.

21. In the `Arrange` section, create an instance of the `ScheduleHelper` class along with a date in the future.

```
    // Arrange
    var now =        new DateTimeOffset(2025, 11, 01, 14, 22, 22, timeZoneOffset);
    var futureDate = new DateTimeOffset(2025, 11, 02, 14, 22, 22, timeZoneOffset);
    var helper = new ScheduleHelper(new FakeSunsetProvider());
    helper.HelperTimeProvider = new FakeTimeProvider(now);
```

22. In the `Act` section, call the `IsInFuture` method.

```
    // Act
    var result = helper.IsInFuture(futureDate);
```

23. In the `Assert` section, make sure the result is "true".

```
    // Assert
    Assert.True(result);
```

24. Here is the completed test.

```
[Fact]
public void IsInFuture_WithFutureDate_ReturnsTrue()
{
    // Arrange
    var now =        new DateTimeOffset(2025, 11, 01, 14, 22, 22, timeZoneOffset);
    var futureDate = new DateTimeOffset(2025, 11, 02, 14, 22, 22, timeZoneOffset);
    var helper = new ScheduleHelper(new FakeSunsetProvider());
    helper.HelperTimeProvider = new FakeTimeProvider(now);

    // Act
    var result = helper.IsInFuture(futureDate);

    // Assert
    Assert.True(result);
}
```

25. Re-run the tests. All tests should pass.

To complete the tests, test a date in the past (should return false) and also what happens if the times are equal (should also return false). You can check the Completed code for examples of these tests.

Wrap Up

In this lab, you were able to use the Property Injection pattern so that the SerialCommander (that uses the hardware) is used by default, but it can be easily replaced for testing. In the final lab, you will see a way to decide what to do based on the application environment. This means that nothing needs to be changed before deployment.

You also saw some of the quirks and issues with testing `DateTime.Now`. Ideally, it is best to avoid statics in tests. Fortunately, we have the `TimeProvider` type to help us out with that.

You saw how to create an abstraction around `DateTime.Now` (or `DateTimeOffset.Now` in this specific case). By using property injection, you get a good default value (the System time), but there is also an injection point to change the value in unit tests.

For the various dependencies, you saw how you can create a fake class that implements an interface that is needed during testing. By using a fake object, you get full control over how an object responds in a test. These are great for dependencies that are incidental to the methods you are testing.

In the next lab, you will use a dependency injection container and learn some techniques for managing dependencies.

END - Lab 02 - Property Injection and Handling DateTime.Now