Lab 02 - Adding Unit Tests

Objectives

In this lab, you will add unit tests for the scheduling features of the project. The current code makes calls to the static `DateTimeOffset.Now` property. This makes unit tests that have to deal with "now" very difficult. First, you will change calls that get the current time so that you can provide your own value for testing. Then you will work with fake and mock objects for the abstractions created in Lab 01. At the end, you will have a set of testing techniques that are transferable to other projects.

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.

> 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. Create a set of unit tests for the Schedule class. The test methods are stubbed out in the `HouseControl.Library.Test` project.
2. Create a set of unit tests for the `ScheduleHelper` class. This class has "Roll" methods to get the next day, weekday, and weekend day. The following methods should have tests:
    - RollForwardToNextDay
    - RollForwardToNextWeekdayDay
    - RollForwardToNextWeekendDay
3. To facilitate the above unit tests, create an abstraction for `DateTimeOffset.Now` that can be set manually for testing.

BONUS

4. Create helper methods for `Today`, `IsInPast`, and `DurationFromNow` along with appropriate tests.

Hints

General

- A unit testing project has been stubbed out using NUnit: `HouseControl.Library.Test`. If you are more comfortable with a different framework (such as xUnit or MSTest), feel free to use that instead.

Unit Testing DateTime.Now

- `DateTimeOffset` is a `DateTime` that also includes timezone information.
- Find all references to `DateTimeOffset.Now` in the code. These should be limited to 2 classes: `Schedule` and `ScheduleHelper`.
- Create an abstraction for providing the current date/time. (Note: You can also investigate the new `TimeProvider` type that was added in .NET 8).
- Use Property Injection to make the time provider available for change in unit tests.

Schedule Tests

- A shell class with unit test methods has already been stubbed out: `HouseControl.Library.Test/ScheduleTests.cs`.
- A file with test data is already set up for the tests to use. The test class has a variable with the location (`ScheduleData`).
- You will need a `SunsetProvider`, however, no values from the SunsetProvider will be used in these tests.
- You can create a ScheduleItem with the type "Once" with the following code:

```
new ScheduleItem(
    1, // device number
    DeviceCommand.On, // command
    new ScheduleInfo()
    {
        EventTime = DateTimeOffset.Now.AddMinutes(-2), // time 2 minutes in the past
        Type = ScheduleType.Once, // schedule type "Once"
    },
    true, // enabled
    "" // schedule set
);
```

ScheduleHelper Tests

- 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 time zone information. You may want to create a field set to the following:

```
TimeZoneInfo.Local.GetUtcOffset(DateTimeOffset.Now())
```

- It may be helpful to use a mocking framework (such as Moq) to create an object to represent the current "fake" time.

- To create a `ScheduleInfo` object for testing, only the `EventTime` (`DateTimeOffset`) and `TimeType` (`ScheduleTimeType`) are required. Default values can be used for the other properties.

- It may be a good idea to have a "TearDown" step that sets the time provider back to its default value. (This is one of the fun things when dealing with statics in tests.)

BONUS "Now" Helper Methods

- Helper methods fit well in the ScheduleHlper class.
- Helper method for `Today` is `Now` with just the date portion. (Note: I would recommend keeping this as a `DateTimeOffset` due to calculations in existing code.)
- Helper method for `IsInPast` compares a DateTimeOffset with `Now` and returns a boolean.
- Helper method for `DurationFromNow` subtracts a `DateTimeOffset` from `Now` and then uses `.Duration()` to return a timespan. Note: `Duration()` returns an absolute value, meaning the result will never be negative.
- Create unit tests for these helper methods.

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 - Unit Testing 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 static helper method on the `ScheduleHelper` class for `Now`.

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

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

```
public static DateTimeOffset Tomorrow()
{
    //return new DateTimeOffset(
```

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

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

```
    //var nextDay = DateTimeOffset.Now.Date.AddDays(1);
    var nextDay = ScheduleHelper.Now().Date.AddDays(1);
```

You should have 3 replacements in `ScheduleHelper`.

> Because the static `Now()` is in the `ScheduleHelper` class, the calls *in this class* could be simplified from `ScheduleHelper.Now()` to just `Now()`.

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

```
public static bool IsInFuture(DateTimeOffset checkTime)
{
    return checkTime > Now();
}
```

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())
```

You should have 3 replacements in `Schedule`.

6. Create an interface named `ITimeProvider` in the `Schedules` folder.

```
namespace HouseControl.Library;

public interface ITimeProvider
{
}
```

Note: You may need to update the namepace to `HouseControl.Library`.

7. Add a `Now` method that returns a `DateTimeOffset`.

```
public interface ITimeProvider
{
    DateTimeOffset Now();
}
```

8. Create a new class called `CurrentTimeProvider` (also in the "Schedules" folder) and implement the interface to call `DateTimeOffset.Now`.

```
namespace HouseControl.Library;

public class CurrentTimeProvider : ITimeProvider
{
    public DateTimeOffset Now()
    {
        return DateTimeOffset.Now;
    }
}
```

9. In the `ScheduleHelper` class, create a new static property to hold the `ITimeProvider`.

```
private static ITimeProvider? timeProvider;
public static ITimeProvider TimeProvider
{
    get { return timeProvider; }
    set { timeProvider = value; }
}
```

10. 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 (timeProvider is null)
        timeProvider = new CurrentTimeProvider();
    return timeProvider;
}
```

> The result is that the the `CurrentTimeProvider` is used by default. So if you do not explicitly set the `TimeProvider` 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`.

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

```
get
{
    return timeProvider ??= new CurrentTimeProvider();
}
```

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

```
private static ITimeProvider? timeProvider;
public static ITimeProvider TimeProvider
{
    get => timeProvider ??= new CurrentTimeProvider();
    set => timeProvider = value;
}
```

13. Update the static `Now` method to use the new property.

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

> Note: if you get an error trying to use `TimeProvider.Now` in this method, it may be because the property is not marked as `static`.

With the abstraction 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 - Schedule Tests

There are a variety of ways to organize tests. For this solution, the test projects match the project under test: `HouseControl.Library.Test` => `HouseControl.Library`. In addition, the test classes match the class under test: `Schedule.Test` => `Schedule`. This is just one approach to organizing tests which works well for this solution.

1. Open the `ScheduleTest.cs` file in the `HouseControl.Library.Test` project.

2. This project is already configured to use NUnit as a testing framework. In addition, a data file used for testing is automatically copied into a folder where the tests have access to it. A field is set to the test file location:

```
readonly string fileName = AppDomain.CurrentDomain.BaseDirectory + "\\ScheduleData";
```

3. Run the existing tests.
   Visual Studio 2022: Open the test explorer from the "Test" menu. Then click the "Run all in View" button on the far left.
   Visual Studio Code: From the command line, navigate to the "HouseControl.Library.Test" folder and type `dotnet test`.

The test results will say either "Not Run" or "Skipped". This is because all of the test stubs have "Inconclusive" results.

3. Look at the first test in the project:

```
[Test]
public void ScheduleItems_OnCreation_IsPopulated()
{
    // Arrange / Act

    // Assert
```

```
    Assert.Inconclusive();
}
```

The tests use a 3 part naming scheme. The first part is the unit under test (`ScheduleItems`). The second is the action run (`OnCreation`). The third is the expected result (`IsPopulated`). So the name of this test tells us that we expect `ScheduleItems` to be populated when the `Schedule` class is created.

> `ScheduleItems` is not a separate property of the `Schedule` class. The `Schedule` class itself is a `List<ScheduleItem>`. So the class itself is a collection of `ScheduleItems`. For example, if you want to know the number of `ScheduleItems`, use `Schedule.Count` (a property on `List<T>`).

In addition, the tests use Arrange / Act / Assert for organization. The Arrange section has any setup code, such as object creation. The Act section is where the action is performed. The Assert section is where the results are checked. In some instances, such as this test, the Arrange and Act sections are combined (since this covers what happens at object creation).

4. Create a new `Schedule` object.

```
// Arrange / Act
var schedule = new Schedule(fileName, );
```

The `Schedule` constructor takes 2 parameters. The first is the data file name (which is in a field). The second parameter is an `ISunsetProvider` (an interface created in Lab 01).

For this set of tests, create a fake sunset provider.

5. In the test project, create a new class called `FakeSunsetProvider`.

```
namespace HouseControl.Library.Test;

public class FakeSunsetProvider
{
}
```

6. Implement in the `ISunsetProvider` interface.

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

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

> Note: you may need to add a using statement for `HouseControl.Sunset` if your IDE does not add it for you.

7. Hardcode some values to return from the 2 methods:

```
public DateTimeOffset GetSunrise(DateTime date)
{
    DateTime time = DateTime.Parse("06:15:00");
    return new DateTimeOffset(date.Date + time.TimeOfDay);
}

public DateTimeOffset GetSunset(DateTime date)
{
    DateTime time = DateTime.Parse("20:25:00");
    return new DateTimeOffset(date.Date + time.TimeOfDay);
}
```

8. Back in the test class, use the `FakeSunsetProvider` to create the `Schedule`.

```
// Arrange / Act
var schedule = new Schedule(fileName, new FakeSunsetProvider());
```

9. In the Assert section, make sure that the `Schedule` has more than one item.

```
// Assert
Assert.That(schedule.Count, Is.GreaterThan(0));
```

> NUnit asserts uses an `Assert.That` syntax. The first parameter is the thing that you are checking (the count of items in the schedule). The second parameter is the condition (greater than 0). There are a number of helper items in NUnit (like `Is`) that is used here. `Is.EqualTo(expectedValue)` is a common condition, but there is quite a bit of flexibility.

10. Review the completed test.

```
[Test]
public void ScheduleItems_OnCreation_IsPopulated()
{
    // Arrange / Act
    var schedule = new Schedule(fileName, new FakeSunsetProvider());

    // Assert
    Assert.That(schedule.Count, Is.GreaterThan(0));
}
```

11. Re-run the tests. There should now be 1 "Passed" test in addition to the inconclusive ones.

12. Review the next test in the class:

```
[Test]
public void ScheduleItems_OnCreation_AreInFuture()
{
    // Arrange / Act

    // Assert
    Assert.Inconclusive();
}
```

When a schedule is loaded from a file, the items are updated to their next future run time. In this test, you will make sure this is true: all `ScheduleItems` have an `EventTime` in the future. The `EventTime` is part of the `ScheduleItem.Info` property.

13. To check that items are in the future, you need the current time. This is part of the arrangement of the test.

```
// Arrange / Act
var schedule = new Schedule(fileName, new FakeSunsetProvider());
var currentTime = DateTimeOffset.Now;
```

The creation of the schedule is the same as the previous test. For the current time, use `DateTimeOffset.Now`. (This test, and other `Schedule` tests, can use the current time; tests for `ScheduleHelper` methods will need a fake time.)

14. For the assert section, loop through the items and check that they are in the future.

```
// Assert
foreach (var item in schedule)
{
```

```
    Assert.That(item.Info.EventTime, Is.GreaterThan(currentTime));
}
```

15. Review at re-run the tests.

```
[Test]
public void ScheduleItems_OnCreation_AreInFuture()
{
    // Arrange / Act
    var schedule = new Schedule(fileName, new FakeSunsetProvider());
    var currentTime = DateTimeOffset.Now;

    // Assert
    foreach (var item in schedule)
    {
        Assert.That(item.Info.EventTime, Is.GreaterThan(currentTime));
    }
}
```

You should now have 2 passing tests.

16. Review the next test:

```
[Test]
public void ScheduleItems_AfterRoll_AreInFuture()
```

For this test, the action is `AfterRoll` -- meaning calling `RollSchedule` on the `Schedule` object. To make sure that items are actually rolled forward, in the arrangement you should set the `EventTimes` to a time in the past.

17. Create the schedule (as in the last 2 tests), then subtract 30 days from all of the `EventTimes`.

```
// Arrange
var schedule = new Schedule(fileName, new FakeSunsetProvider());
var currentTime = DateTimeOffset.Now;
foreach(var item in schedule)
{
    item.Info.EventTime = item.Info.EventTime - TimeSpan.FromDays(30);
    Assert.That(item.Info.EventTime, Is.LessThan(currentTime),
        "Invalid Arrangement");
}
```

The first 2 lines of the Arrange section are the same. Inside the loop, after updating the time, there is a check to make sure it is in the past. This is a "sanity check" to make sure that the arrangement is correct. If this step fails the message "Invalid Arrangement" shows in the test runner so you that this is a setup failure rather than a test failure.

18. For the Act section, call the `RollSchedule` method.

```
// Act
schedule.RollSchedule();
```

19. The Assert section is the same as the previous test: to make sure all items are now in the future:

```
// Assert
foreach (var item in schedule)
{
    Assert.That(item.Info.EventTime, Is.GreaterThan(currentTime));
}
```

20. Review and run the tests.

```
[Test]
public void ScheduleItems_AfterRoll_AreInFuture()
{
    // Arrange
    var schedule = new Schedule(fileName, new FakeSunsetProvider());
    var currentTime = DateTimeOffset.Now;
    foreach(var item in schedule)
    {
        item.Info.EventTime = item.Info.EventTime - TimeSpan.FromDays(30);
        Assert.That(item.Info.EventTime, Is.LessThan(currentTime),
            "Invalid Arrangement");
    }

    // Act
    schedule.RollSchedule();

    // Assert
    foreach (var item in schedule)
    {
        Assert.That(item.Info.EventTime, Is.GreaterThan(currentTime));
    }
}
```

The test runner should now show 3 passing tests.

21. Review the next test:

```
[Test]
public void OneTimeItemInPast_AfterRoll_IsRemoved()
```

This checks an item in the past that is marked as `Once`. See the Hints section for how to create a `ScheduleItem`.

22. Create the schedule (as in previous tests) and add a new item with a past time.

```
// Arrange
var schedule = new Schedule(fileName, new FakeSunsetProvider());

var newItem = new ScheduleItem(
    1,
    DeviceCommand.On,
    new ScheduleInfo()
    {
        EventTime = DateTimeOffset.Now.AddMinutes(-2),
        Type = ScheduleType.Once,
    },
    true,
    ""
);
schedule.Add(newItem);
```

23. Get the original count of items and the count after the new item is added. Then do a sanity check to make sure the item was added successfully.

```
// Arrange
var schedule = new Schedule(fileName, new FakeSunsetProvider());

var originalCount = schedule.Count;

var newItem = new ScheduleItem(...);
schedule.Add(newItem);

var newCount = schedule.Count;
Assert.That(newCount, Is.EqualTo(originalCount + 1),
    "Invalid Arrangement");
```

24. Roll the schedule forward.

```
// Act
schedule.RollSchedule();
```

25. Assert that the count is back to the original count (before adding the item).

```
// Assert
Assert.That(schedule.Count, Is.EqualTo(originalCount));
```

26. Review and run the tests.

```
[Test]
public void OneTimeItemInPast_AfterRoll_IsRemoved()
{
    // Arrange
    var schedule = new Schedule(fileName, new FakeSunsetProvider());
    var originalCount = schedule.Count;

    var newItem = new ScheduleItem(
        1,
        DeviceCommand.On,
        new ScheduleInfo()
        {
            EventTime = DateTimeOffset.Now.AddMinutes(-2),
            Type = ScheduleType.Once,
        },
        true,
        ""
    );
    schedule.Add(newItem);
    var newCount = schedule.Count;
    Assert.That(newCount, Is.EqualTo(originalCount + 1),
        "Invalid Arrangement");

    // Act
    schedule.RollSchedule();

    // Assert
    Assert.That(schedule.Count, Is.EqualTo(originalCount));
}
```

27. Look at the last test in the ScheduleTests class.

```
[Test]
public void OneTimeItemInFuture_AfterRoll_IsStillThere()
```

This test is similar to the last test. The new item is in the future, and the assertion is that it is not removed by the schedule roll.

28. Copy/Paste the previous test. Change the `EventTime` to `+2` instead of `-2`, and change the assertion to check that the schedule count equals the **new count**.

```
[Test]
public void OneTimeItemInFuture_AfterRoll_IsStillThere()
{
    // Arrange
    var schedule = new Schedule(fileName, new FakeSunsetProvider());
    var originalCount = schedule.Count;

    var newItem = new ScheduleItem(
        1,
        DeviceCommand.On,
        new ScheduleInfo()
        {
            EventTime = DateTimeOffset.Now.AddMinutes(+2),
            Type = ScheduleType.Once,
        },
        true,
        ""
    );
    schedule.Add(newItem);
    var newCount = schedule.Count;
    Assert.That(newCount, Is.EqualTo(originalCount + 1),
        "Invalid Arrangement");

    // Act
    schedule.RollSchedule();

    // Assert
    Assert.That(schedule.Count, Is.EqualTo(newCount));
}
```

29. Run all tests. There should now be 5 passing tests.

Step-by-Step - ScheduleHelper Tests

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

2. There is a comment to show which methods to test:

    - `RollForwardToNextDay`
      Rolls a schedule item to the next date after today.
    - `RollForwardToNextWeekdayDay`
      Rolls a schedule item to the next weekday (Monday - Friday) after today.
    - `RollForwardToNextWeekendDay`
      Rolls a schedule item to the next weekend day (Saturday / Sunday) after today.

All methods should have tests for items in the past (which do roll forward) and items in the future (which do not roll forward).

3. For the first test, we will check a Monday date in the past and roll it forward regularly.

```
[Test]
public void MondayItemInPast_OnRollDay_IsTomorrow()
{
}
```

4. In the Arrange section, create a `DateTimeOffset` that happens on a Monday (the date does not matter as long as it is in the past).

```
// Arrange
DateTimeOffset monday = new(2024, 02, 26, 15, 52, 00, );
```

5. To create a `DateTimeOffset`, you need to include a `TimeSpan` that represents the current time difference from UTC. Since this will be used throughout the tests, create a class-level field:

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

The exact value does not matter for our tests. In this case, it represents -2 hour from UTC.

6. Use the `timeZoneOffset` to create the monday variable.

```
DateTimeOffset monday = new(2024, 02, 26, 15, 52, 00, timeZoneOffset);
```

7. Create a variable to hold the expected value. This is tomorrow's date with the time portion from "monday".

```

```

```
DateTimeOffset expected = new(DateTime.Now.AddDays(1) + monday.TimeOfDay,
timeZoneOffset);
```

8. The `RollForwardToNextDay` method needs a `ScheduleInfo` object. Create a new instance. The only values you must set are `EventTime` and `TimeType`. The `TimeType` should be `Standard`.

```
ScheduleInfo info = new()
{
    EventTime = monday,
    TimeType = ScheduleTimeType.Standard,
};
```

9. The last part of the Arrange section is to create the `ScheduleHelper` instance. This needs a sunset provider. Use the `FakeSunsetProvider` created earlier.

```
ScheduleHelper helper = new(new FakeSunsetProvider());
```

10. For the Act and Assert sections, call `RollForwardToNextDay` and compare the result with the `expected` value.

```
// Act
DateTimeOffset actual = helper.RollForwardToNextDay(info);

// Assert
Assert.That(actual, Is.EqualTo(expected));
```

11. Run the test.

At this point, the test most likely will fail. And looking at the message, the expected and actual time zone offsets do not match (unless your computer is set for UTC-2).

The reason for the failure is that this test relies on the `CurrentTimeProvider` that is used by default. You can create your own `ITimeProvider` which gives better control over the tests.

12. Create a new method in the test class called `SetCurrentTime`. This will create a fake time provider and assign it to the `ScheduleHelper`'s `TimeProvider` property.

```
private void SetCurrentTime(DateTimeOffset currentTime)
{
}
```

13. Instead of creating fake object, use the Moq mocking framework. (The Moq NuGet package has already beed added to the unit test project.) A quickstart is available here: https://github.com/devlooped/moq/wiki/Quickstart.

14. Mocking frameworks work by creating in-memory objects that you configure with the behavior that you want. The advantage of using a mock over a fake is that you only need to provide behavior that you need. For a complex object, this may mean only filling a couple of properties or methods rather than creating an entire class.

15. Create a new mock on the `ITimeProvider` interface.

```
var mockTime = new Mock<ITimeProvider>();
```

16. Configure the mock to return the `currentTime` parameter for the `Now()` method. Check the quickstart if you want all of the options.

```
mockTime.Setup(t => t.Now()).Returns(currentTime);
```

This code calls `Setup` on the mock object. The lambda expression denotes the setup is for the `Now()` method. The `.Returns` configures the mock object to return the `currentTime` value if anyone calls the `Now()` method on this mock object.

17. The last step is to assign the mock to the `TimeProvider` property. To get to the actual `ITimeProvider` of the mock, use the `Object` property.

```
ScheduleHelper.TimeProvider = mockTime.Object;
```

Here is the entire method:

```
private void SetCurrentTime(DateTimeOffset currentTime)
{
    var mockTime = new Mock<ITimeProvider>();
    mockTime.Setup(t => t.Now()).Returns(currentTime);
    ScheduleHelper.TimeProvider = mockTime.Object;
}
```

18. Back in the test, create an `DateTimeOffset` for a Thursday (it doesn't matter which one, as long as it is after the `monday` date). Use this value to set the current time.

```
DateTimeOffset thursday = new(2024, 02, 29, 16, 35, 22, timeZoneOffset);
SetCurrentTime(thursday);
```

19. Update the `expected` value to use the `thursday` value. Here is the updated Arrange section.

```
// Arrange
DateTimeOffset thursday = new(2024, 02, 29, 16, 35, 22, timeZoneOffset);
SetCurrentTime(thursday);

DateTimeOffset monday = new(2024, 02, 26, 15, 52, 00, timeZoneOffset);
DateTimeOffset expected = new(thursday.Date.AddDays(1) + monday.TimeOfDay,
timeZoneOffset);

ScheduleInfo info = new()
{
    EventTime = monday,
    TimeType = ScheduleTimeType.Standard,
};

ScheduleHelper helper = new(new FakeSunsetProvider());
```

20. Rerun the test. It should now pass.

21. **But there's a problem**:
    Some of the `Schedule` tests are now failing.

This is because the default time provider is needed by the `Schedule` tests, and that default time provider has been replaced.

> This is one of the problems of unit testing static objects. They are potentially shared across all of the tests. To be a good steward, when you override a default, you should put it back after you are done.

22. Create a `TearDown` method in the `ScheduleHelper.Tests` class to reset the `TimeProvider`. This will automatically run after every test. In NUnit a tear down method is denoted with the attribute `[TearDown]`.

```
[TearDown]
public void Teardown()
{
    // Reset TimeProvider to the default that uses the real time
    ScheduleHelper.TimeProvider = new CurrentTimeProvider();
}
```

Another option is to set the `TimeProvider` property to "null". The default value will get used the next time it is accessed.

23. Rerun the tests. The schedule tests should now pass again.

Since the `Schedule` tests need the default time provider. You may want to add a setup method in the `Schedule.Test` class to reset the `TimeProvider` to default. A setup method runs before each test.

```
[SetUp]
public void Setup()
{
    // Reset TimeProvider to the default that uses the real time
    ScheduleHelper.TimeProvider = null;
}
```

> One restriction when dealing testing with static object is that the tests cannot be run in parallel. Most testing frameworks default to running tests one at a time (although the order is not necessarily deterministic). Many testing frameworks also support parallel test runs to speed things up. When using statics in tests, this can lead to instability and unexpected static values. **When testing statics, it is best to run tests one at a time.**

24. The next test is for a Monday in the future. When the `RollForwardToNextDay` method is called, the time should be unchanged.

```
[Test]
public void MondayItemInFuture_OnRollDay_IsUnchanged()
{
}
```

25. The internals of this test are almost identical to the previous. Here are the differences.
    - The `monday` date should be a date after the `thursday` value.
    - The `expected` value is `monday`.

```
[Test]
public void MondayItemInFuture_OnRollDay_IsUnchanged()
{
    // Arrange
    DateTimeOffset thursday = new(2024, 02, 29, 16, 35, 22, timeZoneOffset);
    SetCurrentTime(thursday);
    DateTimeOffset monday = new(2024, 03, 04, 15, 52, 00, timeZoneOffset);
    DateTimeOffset expected = monday;

    ScheduleInfo info = new()
    {
```

```
        EventTime = monday,
        TimeType = ScheduleTimeType.Standard,
    };

    ScheduleHelper helper = new(new FakeSunsetProvider());

    // Act
    DateTimeOffset actual = helper.RollForwardToNextDay(info);

    // Assert
    Assert.That(actual, Is.EqualTo(expected));
}
```

26. To check the weekday roll, set the current time to a Friday, set the Monday value to a date in the "past". And set expected to use Friday plus 3 days (i.e., the following Monday). Here is the test. This is similar to the previous tests, however make sure that you call the `helper.RollForwardToNextWeekdayDay()` method.

```
[Test]
public void MondayItemInPastAndTodayFriday_OnRollWeekdayDay_IsNextMonday()
{
    // Arrange
    DateTimeOffset friday = new(2024, 03, 01, 16, 35, 22, timeZoneOffset);
    SetCurrentTime(friday);
    DateTimeOffset monday = new(2024, 02, 26, 15, 52, 00, timeZoneOffset);
    DateTimeOffset expected = new(friday.Date.AddDays(3) + monday.TimeOfDay,
timeZoneOffset);
    ScheduleInfo info = new()
    {
        EventTime = monday,
        TimeType = ScheduleTimeType.Standard,
    };
    ScheduleHelper helper = new(new FakeSunsetProvider());

    // Act
    DateTimeOffset actual = helper.RollForwardToNextWeekdayDay(info);

    // Assert
    Assert.That(actual, Is.EqualTo(expected));
}
```

27. Now test the same as the previous test with Monday in the "future" (meaning, after the Friday date). The expected value is that it is unchanged.

```
[Test]
public void MondayItemInFutureAndTodayFriday_OnRollWeekdayDay_IsUnchanged()
{
    // Arrange
    DateTimeOffset friday = new(2024, 03, 01, 16, 35, 22, timeZoneOffset);
    SetCurrentTime(friday);

    DateTimeOffset monday = new(2024, 03, 04, 15, 52, 00, timeZoneOffset);
    DateTimeOffset expected = monday;
    ScheduleInfo info = new()
    {
        EventTime = monday,
        TimeType = ScheduleTimeType.Standard,
    };
    ScheduleHelper helper = new(new FakeSunsetProvider());

    // Act
    DateTimeOffset actual = helper.RollForwardToNextDay(info);

    // Assert
    Assert.That(actual, Is.EqualTo(expected));
}
```

28. Create the same pair of tests for `RollForwardToNextWeekendDay`. Use a Friday for the current date, Saturday for the test date (one in the "past", one in the "future").

```
[Test]
public void SundayItemInPastAndTodayThursday_OnRollWeekendDay_IsSaturday()
{
    // Arrange
    DateTimeOffset thursday = new(2024, 02, 29, 16, 35, 22, timeZoneOffset);
    SetCurrentTime(thursday);

    DateTimeOffset sunday = new(2024, 02, 25, 15, 52, 00, timeZoneOffset);
    DateTimeOffset expected = new(thursday.Date.AddDays(2) + sunday.TimeOfDay,
timeZoneOffset);
    ScheduleInfo info = new()
    {
        EventTime = sunday,
        TimeType = ScheduleTimeType.Standard,
    };
    ScheduleHelper helper = new(new FakeSunsetProvider());

    // Act
    DateTimeOffset actual = helper.RollForwardToNextWeekendDay(info);

    // Assert
```

```
    Assert.That(actual, Is.EqualTo(expected));
}

[Test]
public void SaturdayItemInFutureAndTodayFriday_OnRollWeekendDay_IsUnchanged()
{
    // Arrange
    DateTimeOffset friday = new(2024, 03, 01, 16, 35, 22, timeZoneOffset);
    SetCurrentTime(friday);

    DateTimeOffset saturday = new(2024, 03, 02, 15, 52, 00, timeZoneOffset);
    DateTimeOffset expected = saturday;
    ScheduleInfo info = new()
    {
        EventTime = saturday,
        TimeType = ScheduleTimeType.Standard,
    };
    ScheduleHelper helper = new(new FakeSunsetProvider());

    // Act
    DateTimeOffset actual = helper.RollForwardToNextWeekendDay(info);

    // Assert
    Assert.That(actual, Is.EqualTo(expected));
}
```

29. As you can see, once you get the initial tests laid out, it is pretty easy to add more scenarios.

Depending on the tests, it may make sense to parameterize the tests rather than creating individual tests. With parameterized tests, you can pass in different sets of parameters to the same test method. You can check parameterization for your choses unit test framework to learn more.

Step-by-Step - "Now" Helper Methods

As a bonus, you can do a bit of refactoring and testing of "Now" helper methods. The code currently has 3 static helper methods:

- Now()
- Tomorrow()
- IsInFuture(DateTimeOffset)

We will create tests for these items as well as add a few more helpers to make code more readable:

- Today()
- IsInPast(DateTimeOffset)
- DurationFromNow(DateTimeOffset)

For this section, all `static` methods go in the `ScheduleHelper` class. All tests go in the `ScheduleHelper.Tests` class.

1. Create a test for `Now()`.

```
[Test]
public void Now_ReturnsConfiguredTime()
{
    // Arrange
    DateTimeOffset thursday = new(2024, 02, 29, 16, 35, 22, timeZoneOffset);
    SetCurrentTime(thursday);
    DateTimeOffset expected = thursday;

    // Act
    var actual = ScheduleHelper.Now();

    // Assert
    Assert.That(actual, Is.EqualTo(expected));
}
```

By now, you should be able to see how this test works. First, it sets the current time to "Thursday". Then it calls the `Now()` method and makes sure the result is "Thursday".

2. Create a test for `Tomorrow`. This will be similar to the test for `Now`.

```
[Test]
public void Tomorrow_ReturnsConfiguredDatePlusOne()
{
    // Arrange
    DateTimeOffset thursday = new(2024, 02, 29, 16, 35, 22, timeZoneOffset);
    SetCurrentTime(thursday);
    DateTimeOffset expected = new(thursday.Date.AddDays(1), timeZoneOffset);

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

    // Assert
    Assert.That(actual, Is.EqualTo(expected));
}
```

The expected value is just the date portion + 1 day, with the current time zone offset.

3. Create a new static method for `Today` which is just the Date portion from `Now` (along with the current time zone offset).

```
public static DateTimeOffset Today()
{
    return new DateTimeOffset(Now().Date, Now().Offset);
}
```

4. Create a unit test for `Today` (similar to the test for `Now`).

```
[Test]
public void Today_ReturnsConfiguredDate()
{
    // Arrange
    DateTimeOffset thursday = new(2024, 02, 29, 16, 35, 22, timeZoneOffset);
    SetCurrentTime(thursday);
    DateTimeOffset expected = new DateTimeOffset(thursday.Date, timeZoneOffset);

    // Act
    var actual = ScheduleHelper.Today();

    // Assert
    Assert.That(actual, Is.EqualTo(expected));
}
```

5. Refactor calls to `ScheduleHelper.Now().Date`. In the `Schedule` class, `Now().Date` is used in the `LoadSchedule` method.

```
public void LoadSchedule()
{
    this.Clear();
    this.AddRange(Loader.LoadScheduleItems(filename));

    // update loaded schedule dates to today
    //DateTimeOffset today = ScheduleHelper.Now().Date;
    DateTimeOffset today = ScheduleHelper.Today();
    foreach (var item in this)
    {
        item.Info.EventTime = today + item.Info.EventTime.TimeOfDay;
    }
    RollSchedule();
}
```

6. Run the unit tests.

This tells us 2 things. First, our new method works. Second, that our refactoring did not break any existing functionality. To me, the second factor is really important since it lets me move forward without worrying about

existing code that has tests.

7. Create unit tests for the `IsInFuture` method using both cases: parameter is in the past, parameter is in the future.

```
[Test]
public void PastDate_IsInFuture_ReturnsFalse()
{
    // Arrange
    DateTimeOffset thursday = new(2024, 02, 29, 16, 35, 22, timeZoneOffset);
    SetCurrentTime(thursday);
    DateTimeOffset monday = new(2024, 02, 26, 15, 52, 00, timeZoneOffset);

    // Act
    var actual = ScheduleHelper.IsInFuture(monday);

    // Assert
    Assert.That(actual, Is.False);
}

[Test]
public void FutureDate_IsInFuture_ReturnsTrue()
{
    // Arrange
    DateTimeOffset thursday = new(2024, 02, 29, 16, 35, 22, timeZoneOffset);
    SetCurrentTime(thursday);
    DateTimeOffset monday = new(2024, 03, 04, 15, 52, 00, timeZoneOffset);

    // Act
    var actual = ScheduleHelper.IsInFuture(monday);

    // Assert
    Assert.That(actual, Is.True);
}
```

8. Run the tests.

9. Create a static `IsInPast` method that compares a parameter date to `Now()`.

```
public static bool IsInPast(DateTimeOffset checkTime)
{
    return checkTime < Now();
}
```

10. Create unit tests for both cases (parameter is in the past, parameter is in the future). These are similar to the `IsInFuture` tests.

```
[Test]
public void PastDate_IsInPast_ReturnsTrue()
{
    // Arrange
    DateTimeOffset thursday = new(2024, 02, 29, 16, 35, 22, timeZoneOffset);
    SetCurrentTime(thursday);
    DateTimeOffset monday = new(2024, 02, 26, 15, 52, 00, timeZoneOffset);

    // Act
    var actual = ScheduleHelper.IsInPast(monday);

    // Assert
    Assert.That(actual, Is.True);
}

[Test]
public void FutureDate_IsInPast_ReturnsFalse()
{
    // Arrange
    DateTimeOffset thursday = new(2024, 02, 29, 16, 35, 22, timeZoneOffset);
    SetCurrentTime(thursday);
    DateTimeOffset monday = new(2024, 03, 04, 15, 52, 00, timeZoneOffset);

    // Act
    var actual = ScheduleHelper.IsInPast(monday);

    // Assert
    Assert.That(actual, Is.False);
}
```

Note that "Monday" has different dates in these two tests. The rest is the same (other than the assertion).

11. Refactor the `Schedule` class to use `IsInPast`. This is in the `RollSchedule` method.

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

This may seem like a trivial refactoring, but it shows intent. Rather than a comparison, the code asks whether an item is in the past.

12. Rerun the tests. This confirms that we did not change existing functionality.

13. Tie `IsInPast` and `IsInFuture` together, meaning, have one method call the other. This will ensure that the code is always synchronized. Here is an update to `IsInFuture`:

```
public static bool IsInFuture(DateTimeOffset checkTime)
{
    return !IsInPast(checkTime);
}
```

14. Rerun the tests.

15. Create a method for `DurationFromNow` that returns a `TimeSpan`. For this, you can subtract the dates which results in a `TimeSpan`. `TimeSpan` has a `Duration` method that will give the absolute value back (meaning, you do not need to worry about negative values).

```
public static TimeSpan DurationFromNow(DateTimeOffset checkTime)
{
    return (checkTime - Now()).Duration();
}
```

16. Create a unit test for `DurationFromNow`. This can set the current time, and then set a time to current time + 10 minutes. Duration should be that same 10 minute `TimeSpan`.

```
[Test]
public void Time10MinutesInPast_DurationFromNow_Returns10Minutes()
{
    // Arrange
    DateTimeOffset thursday = new(2024, 02, 29, 16, 35, 22, timeZoneOffset);
    SetCurrentTime(thursday);
    DateTimeOffset testTime = thursday.AddMinutes(-10);
    TimeSpan expected = new(0, 10, 0);

    // Act
    var actual = ScheduleHelper.DurationFromNow(testTime);

    // Assert
    Assert.That(actual, Is.EqualTo(expected));
}
```

17. Refactor the Schedule class to use the new method. This is needed in the `GetCurrentScheduleItems` method. This method looks for items that are within 30 seconds of the current time so that it can execute the commands.

```
// Updated Method
public List<ScheduleItem> GetCurrentScheduleItems()
{
```

```
    return this.Where(si => si.IsEnabled &&
        ScheduleHelper.DurationFromNow(si.Info.EventTime) <
TimeSpan.FromSeconds(30))
        .ToList();
}
```

18. Rerun the tests.

> `DurationFromNow`, `IsInPast`, and `IsInFuture` are all interesting candidates for extension methods. In order to do this, the methods would need to be moved to a static class, and the `this` keyword added before the parameters. Since it relies on the `TimeProvider`, some other methods would need to be moved around as well.

Wrap Up

This this lab, you saw some of the quirks and issues with testing static methods. Ideally, it is best to avoid statics in tests, but there are some areas (such as `DateTime.Now`) that are difficult to avoid.

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, 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 addition to fake objects, you also used a mocking framework (Moq) to create an in-memory test object. These are good particularly for complex objects because you only need to configure the methods, properties, and other members that you care about.

Hopefully you are more comfortable with unit testing and see the importance of the abstractions that you created in Lab 01.

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

END - Lab 02 - Adding Unit Tests