# C#: Using Defensive Programming to Refactor

Eric J Fisher
AUTHOR, PLURALSIGHT
@EricJamesFisher

# What is Defensive Programming?

Defensive programming is an approach to coding focused on improving code quality and reducing bugs.

# Defensive Programming is

| | |
|---|---|
| Clean Code | Code that is written in such a manner that it is easy to read and modify. |
| Predictable | Behaves consistently doing what is expected and not doing what is not expected. |
| Testable | Code should can be easily tested covering a variety of scenarios and that testing should be automatable. |

*Note, we will only be covering the basics of unit testing, once you complete this course if you'd like to learn more about testing I encourage you to take a course specifically on testing!*

# Consider this code

Is this code clean?

Example.cs

```
Main()
{
  var input = Console.ReadLine();
  string appointment = File.Read("file.txt").Last().Split(' ')[0];
  appointment.Time + 15;
  if(appointment.Hour > 17) {
  appointment.Day + 1, Hour = 9; }
  appointment2 = input + appointment;
  File.Write("file.txt",appointment2);
}
```

*Is it clear what this method and it's variables are used for based on their names?*

*Is this code easy to read and understand?*

*Is this code consistent and easy to modify?*

# Consider this code

## This code is not clean

Example.cs

```
Main()
{
  var input = Console.ReadLine();
  string appointment = File.Read("file.txt").Last().Split(' ')[0];
  appointment.Time + 15;
  if(appointment.Hour > 17) {
  appointment.Day + 1, Hour = 9; }
  appointment2 = input + appointment;
  File.Write("file.txt",appointment2);
}
```

> "Main" doesn't provide any indication what it'll do.

> The condition in this method has odd spacing that makes it harder to read.

> This code is inconsistent in styles of variable declaration and spacing.

# Our Method Does A Lot of Different Things

This code appears to be doing four different things

Example.cs

```
Main()
{
  var input = Console.ReadLine();
  string appointment = File.Read("file.txt").Last().Split(' ')[0];
  appointment.Time + 15;
  if(appointment.Hour > 17) {
  appointment.Day + 1, Hour = 9; }
  appointment2 = input + appointment;
  File.Write("file.txt",appointment2);
}
```

*Get input from the user*

*Get the last scheduled appointment*

*Get the next available appointment time*

*Save the new appointment to the schedule*

*This makes code less clean, predictable, and testable. Let's refactor these functions into separate methods*

# Cleaning Up Our Code

We move each function into it's own method under a newly created Appointment class, since they're all dealing with appointments.

Example.cs

```
Main()
{
  var input = Console.ReadLine();
  Appointment.ScheduleAppointment(input);
}
```

Appointment.cs

```
public void ScheduleAppointment {…}

public DateTime GetLast {…}

public DateTime GetNextAvailable {…}

public void Save {…}
```

# Is This Predictable?

Is it clear what it will do? Is it doing what is expected? Is it doing anything unexpected?

Appointment.cs

```
public string GetLast()
{
  return File.Read("file.txt").Last().Split(' ')[0];
}
```

*Could you predict what this method does based on it's name?*

*Is an error when the file doesn't exist expected behavior?*

*What if the file is empty, or not formatted correctly? Should we still return the string?*

# Making Our Code Predictable

We can make our code more predictable using guard code

Appointment.cs

```
DateTime GetLast()
{
  if(!File.Exists("file.txt"))
    File.Create("file.txt");

  string dateString = File.Read("file.txt")?.Last()?.Split(' ')[0];

  DateTime appointment;
  if(!DateTime.TryParse(out appointment, dateString))
    appointment = DateTime.Now;

  return appointment;
}
```

*We can add guard code to create the file if it's not found*

*We can prevent exceptions if the file was empty*

*And we can handle if the file wasn't formatted correctly*

# What About Testability?

GetNextAvailable is easily tested now, but the other methods would still require more work to be testable

Example.cs

```
Main()
{
  var input = Console.ReadLine();
  Appointment.ScheduleAppointment(input);
}
```

*Main, ScheduleAppointment, GetLast, and Save depend on External resources so aren't easy to unit test*

*GetNextAvailable doesn't use external resources though so should be easy to test, let's start there*

Appointment.cs

```
public void ScheduleAppointment {…}

public DateTime GetLast {…}

public DateTime GetNextAvailable {…}

public void Save {…}
```

# Planning Our First Unit Test

For our first unit test we should figure out what one thing we're testing

Appointment.cs

```
Public DateTime GetNextAvailable(dateTime lastAppointment)
{
  DateTime nextAppointment = lastAppointment;
  nextAppointment.AddMinutes(15);
  if(nextAppointment.Hours > 17)
  {
    nextAppointment.AddDays(1);
    nextAppointment.Time = 9;
  }
  return nextAppointment;
}
```

*Our method has two expected scenarios*
*- the condition isn't met and our results are 15 minutes after the lastAppointment time*
*- the condition is met and our results are 9am the day after the lastAppointment*

# First Unit Test

First we'll test the results for when the condition isn't met

AppointmentTests.cs

```
GetNextAvailable_BeforeEndOfDay_FifteenMinutesLater()
{
    // Arrange
    var lastAppointment = new DateTime(2000,1,1,10,0,0);
    var expected = lastAppointment.AddMinutes(15);

    // Act
    var actual = GetNextAvailable(lastAppointment);

    // Assert
    Assert.True(expected == actual)
}
```

*There are many strategies for naming tests, I personally lean toward MethodBeingTested_WhatSituation_ExpectedResults*

*First, we'll prep what's needed to run our test we'll provide an argument with a time of 10 to not trip the condition, we'll also create a variable of what results we are expecting at the end*

# First Unit Test

First we'll test the results for when the condition isn't met

AppointmentTests.cs

```
GetNextAvailable_BeforeEndOfDay_FifteenMinutesLater()
{
    // Arrange
    var lastAppointment = new DateTime
    var expected = lastAppointment.Add

    // Act
    var actual = GetNextAvailable(lastAppointment);

    // Assert
    Assert.True(expected == actual);
}
```

*Next we'll actually run the method we're testing and get the results*

*And finally we'll Assert the results were what we expected. If the results don't match, our test will fail letting us know something is wrong.*

# Test All The Things

We should test all the expected behaviors of our method

Appointment.cs

```
Public DateTime GetNextAvailable(dateTime lastAppointment)
{
  DateTime nextAppointment = lastAppointment;
  nextAppointment.AddMinutes(15);
  if(nextAppointment.Hours > 17)
  {
    nextAppointment.AddDays(1);
    nextAppointment.Time = 9;
  }
  return nextAppointment;
}
```

*Our first test won't cover when adding 15 minutes makes our Hours more than 17, so we would want to test this as well.*

*Generally you want test all code paths of your method, and predictable situations your method might face. These tests can be automated in a variety of ways so would pass as "Testable".*

# Summary

Defensive programming is an approach to coding with a focus on improving code quality and reducing bugs

Defensive Code is Clean, Predictable, and Testable

Clean Code is easy to read and modify

Predictable code consistently does what you'd expect, and doesn't do what you wouldn't expect

Testable code should be able to be tested automatically

You should have a test for every code path of your methods.