Project Overview

For these lessons we will be working with the code that is in the lesson folder. We have created a sample application that is representative of the kind of work the each of us does every day but is still simple enough to be able to complete within a workshop. It loosely follows the example that we went through together during the demo, but this time it is giving you a chance to try it out and apply what we learned.

Data: There are three CSV files that will be needed for this (Teachers.csv, Students.csv, Assignments.csv) all of which are in the data folder under the lessons.

Solutions: Within each lesson there is a ***starting*** folder and a ***solution*** folder. We wanted to make sure that you had an example of what we thought a proper solution would like at the end of the lesson.

Questions: Please feel free to ask questions on anything that you are unsure of or that doesn't make sense. This workshop is about you, and we are here to help.

Lesson 1. Code Discovery

Objective: Look at the code, list what it is doing, and start thinking about how it can be refactored into clean code.

Learning Objective: We all will inherit code that was written by others, some of it will be good and others will be bad. So, it becomes a critical skill to assess what we are getting into, what patterns were established, and how deep the refactoring rabbit hole goes. Failure to do this means that you could inadvertently break something downstream or end up refactoring the entire application, just to fix one class. Also, it is more important to follow existing patterns with the code (even if they are 10 years old) then to mix old and new patterns together.

Initial State:

- One giant method that performs action of the application.
- No validation or type safety around input.
- All calculations are performed in-line, so they can't re-used or easily changed
- No project organization or greater architectural design

Desired State:

- Same as before we started since we aren't yet making any changes at this point.

Lesson 2. Setup New Structure and Refactor into it

Objective: Now that we have a game plan, it's time to start implementing it. We will need to create our different layers, refactor our current implementation into these new layers, and create tests around these new implementations.

Learning Objective: The idea is to split apart large classes and methods into individual components so that we don't violate the Single Responsibility Principle and do so in a way that is more grouped and organized. This is where we really start creating those LEGO pieces that we will use to compose our application.

Initial State:

- Same as Lesson 1, since we didn't change anything

Desired State:

- New projects created for the Domain and Contract layers
- All processing / business logic removed from the Console layer, so that it can serve as a pass-through orchestrator only
- No giant classes or methods (no method longer than 25 lines)
- We should have many more classes and files now
- Tests written around the critical parts such as serializers and calculators (not around POCOs)
- Project References should only be the following:
    - Console references Contracts and Domain
    - Domain references Contracts
    - Contracts references nothing

Remarks:

It is important to create the proper structure to store our new code. This will set the foundation and will allow everything else to fall in line. Failure to do this will lead us to replacing bad code with other bad code.

We will start by creating new projects to house the different parts of the app. This is known as n-Tier architecture. In our case, we will have 4 tiers. The first is the ***Console***, which acts as the starting point of our application, as well as the orchestration for what the application will do. All the code in this layer will either be specific to the console itself or a pass-through into our actual business logic. The second layer is the ***Domain***, where we will store all our business logic, such as calculators, extension methods, etc. that are specific to the application itself. Third is ***Domain.Tests***, where we will put all the tests for the Domain project. The last layer is our ***Contracts*** where we will store our data transfer classes (POCOs) and common interfaces.

Contracts are put into a separate layer, so that they can easily be shared at any level of the application. This prevents us from having to create my Domain version of the Console's data class. Thus, removing the need for unnecessary mapping code.

Note, that if it is generic enough to be re-used across applications, then it should go into its own project (layer), so later it can easily be extracted into a NuGet package.

Tests are always created in separate projects to keep them isolated and so that we don't deploy them.

Lesson 3. Refactor Manual CSV Parsing to use a library

Objective: We want to get rid of our manual CSV parsing, and instead use CsvHelper.

Learning Objective: Sometimes when we write code it is easy to forget that libraries exist, and we instead end up inventing our own wheels all over again. This can be bad since it means we must do a lot more work, instead of standing on the shoulders of giants. It also, means that our application won't be as robust or reusable, because we are focused on how to get the records from a CSV file so that we can perform that actual task of calculating grades, instead of our task being, to create a robust library to read CSV files following the specification.

Initial State:

- We have a solution with proper layers, so that everything has its place
- No big classes or methods, since everything has been split out and refactored
- Still manually parsing CSV file
- No type validation or safety around the CSV parsing

Desired State:

- All of our manual CSV parsing code has been removed
- CSV parsing should be a single line of code within our orchestrator (e.g. Program.cs)
- We are using CsvHelper to do the heaving lifting for us

Remarks:

Doing this serves several benefits. First is that we don't have to be well versed in the CSV specification, so that we can handle unique cases such as a column that contains a comma or spans multiple lines. Second, is that this library is already tested, so it should be robust enough to handle whatever we can throw at it. Lastly, it reduces the lines of code that we need to write. The less code we write, the less chance for errors, and the faster we can write it. Using a library like this can push us toward proper architecture if we aren't already, since it requires us to have POCO classes created so it can do the conversion for us, as opposed to just doing a string split and referencing the column by its index. This makes our code more robust since we are using names not indexes. So for example, if you added a new column between index 3 and 4, everything would still work properly (including the new column) instead of everything worked up till that index, and everything after broke (creating an off-by-one index error).

Look at the project we did during the demo (…\src\SoftwareDoneProperly\Reusable\SharedCsv\CsvSerializer.cs). It has the code for the generic CsvSerializer.

Lesson 4. Dependency Injection

Objective: We want to better architect our code so that it is easy to maintain and easily can handle changing business requirements. We accomplish this using dependency injection so that we better separate our codebase and make it easier to do that.

Learning Objective: As the name implies, we want to inject any dependency that our class needs into the constructor of that class. There is a saying in DI, that says NEW = GLUE, and anytime you new up a class that you depend on for some functionality, then you are tightly coupling that implementation to the class that needs it. In addition, if we were to write tests, then the only tests we can create write are integration tests.

Dependency injection is accomplished with the use of interfaces. Interfaces act as placeholders so that we know what methods and properties the implementing class will have, but not the specific logic itself. This allows us to easily change out the implementation for different parts of our application. For example, let's say that we have a class called TeacherRetriever, whose sole purpose it is to retrieve teachers, be it from a file, database, or web service. The second we choose one, we are stuck with it.

For example, choosing to work with files means our application is solely relying on a file system, and that we can't easily handle other forms such as database. This also means that our Unit Tests become Integration Tests, which we don't want. Instead, we create an interface called ITeacherRetriever (note that interfaces always start with the letter I), which has one method Retrieve (IList<Teacher> Retrieve()). Then I can create several classes that inherit this interface. One could be FileTeacherRetriever which reads from a file, InMemoryTeacherRetreiver which works with in-memory collections (super helpful when creating unit or end-to-end unit tests), or DatabaseTeacherRetriever which reads from the database.

Now, when I want to use this, say in my TeacherGradeCalculator class, I just inject ITeacherRetriever and call the Retrieve() method to get my teachers. In this class I don't care where the teachers come from, I just need them to do the work that I need. After this, at the start of my application (composition time), I specify which implementation I would like to use for ITeacherRetriever, and it will automatically be used for me in my TeacherGradeCalculator. This is handy since I can use DatabaseTeacherRetriever for my application, InMemoryTeacherRetriever for unit tests, and FileTeacherRetriever for my integration tests. Later, say we need add a WebServiceTeacherRetriever, then we can do easily do that by creating this new class, write tests around it, and replace it in our bindings (how we map FileTeacherRetriever to ITeacherRetriever), and we are done. The rest of our application will just work, since we abide by the interface. We can ensure this by running all of our tests and our application will tell us.

Another important concept that we need to talk about is Inversion of Control (IoC). As the name implies, we are inverting the control of object creation over to something else. So instead of newing up an object, we tell a container how we want this object created and let it do it for us. It creates, maintains, and disposes of that object for us. All we need to do is inject our dependencies and it will automatically give us the instance we need, based on the configuration we told it. There are many containers out there, but this is now a built-in mechanism within .Net.

Initial State:

- A usable, yet tightly couple application

Desired State:

- Interfaces around classes that we want to inject
- Wire up the interface to class bindings in a Inversion of Control Container.
- No new() for anything that our class depends on, except for POCOs
- Cleaned up tests so we can truly have unit and integration tests

Remarks:

DI and IoC are fundamental concepts to create great applications. It promotes industry best practices such as Single Responsibility Principle, proper architecture, loose coupling, and easy adapting changing business requirements.

While there are other forms and implementations of Dependency injection (DI), constructor injection is the one that we want to focus on since it is the one we will be using most often. Some other forms such as property injection are harder to implement and very easily lead to anti-patterns and other code smells, that made the code hard to rationalize about.

Lesson 5. Code Assessment

Objective: We want to stand back and look at our application and see what we have completed and what we still need to refactor/make better.

Learning Objective: Discuss what we could improve and what we are going to improve. We must remember that we are working against deadlines and something, while may seem like a good idea, might not provide any business value, and delay a deliverable.

Initial State:

- A completed application that we can be proud and confident to deploy to production, while still being easily readable and maintainable, and easily adaptable to changing business requirements

Desired State:

- Discussion on what improvements we could make, and which ones would be a good idea, and which ones wouldn't. Some examples are:
    - What classes should be static and which ones shouldn't be?
    - Should we create an orchestrator/processor class, so we can test without the complexity of the console application?
    - Creating a NuGet package for our CsvSerializer?
    - What can be made more generic/reusable?
    - Anything else you can come up with

Remarks:

This goes back to being a Pragmatist not a Purist. We know that we aren't going to be able to write 100% perfect code every time, but that we feel confident with where we are with the app, and that it is going to work the way we want in production. This is how we know we are at a good stopping point, until new requirements are introduced. If you were to continue, then you run the risk of bike shedding (endlessly refactoring without making any improvement) and just waste your time. It is ok to take a little technical debt, we just want to manage it. For example, we created a generic CsvSerializer, but don't turn it into a NuGet package, until we need it on another project. This follows the YAGNI (You Ain't Gonna Need It) principle until it is actually needed. At which point, it would be removed from the application it was created in, be placed in its own repository, added to the company's private NuGet feed, and shared into other applications that need it.