Refactoring C# to F# in Action

Introduction

As part of this demonstration of refactoring C# to F#, I've decided to utilise C# 9.0 functionality. This illustrates how C# continues to adopt F# functionality and much of the time the code is almost indistinguishable from the F# variant. However during my time experimenting with C# 9.0, pitfalls became pretty apparent. C# 9.0 simply isn't supported on many legacy projects so it is necessary to forego in the event where you need to support older projects. Meanwhile F# is backward compatible, you might not always get the latest and greatest features if supporting particularly old .Net Framework projects however it is possible to get a nice balance.

The sample project I have introduced is a basic enough implementation of a Book with some functionality that allows for validation and the execution of some basic business logic. I have also included included unit tests that verify the logic.

At times, there were issues around F# where things didn't behave as we expected. Hopefully this piece will help other developers to bypass these issues entirely. This acts as a follow up piece to this article on refactoring tips.

Records

Records are effectively the F# equivalent of a class except they are immutable. C# 9.0 also introduced immutable records.

C# Record

```
public record Book
{
    public string Isbn { get; init; }
    public string Name { get; init; }
    public string Author { get; init; }
    public int PageCount { get; init; }
    public string Publisher { get; init; }
    public List<Genre> Genre { get; init; }
    public string NextInSeries { get; init; }
    public int? Rating { get; init; }
}
```

The key difference between a class and a record in terms of defining them is the use of `record` instead of `class`. And the introduction of `init` in addition to standard getters and setters. The init prevents the modification of a field after the creation of a record.

F# Record

```
type BookV2 =
{
    Isbn: string
    Name: string
    Author: string
    PageCount: int
    Publisher: string
    Genre: List<GenreV2>
    NextInSeries: Option<string>
    Rating: Option<int>
}
```

The F# equivalent record has a few minor differences. The composition being the first example, there's less boilerplate. The other most apparent difference is the use of `Option`. This is similar to a nullable value, if null the type will be `None` and if populated it will be boxed up as `Some` with the value inside it. This allows for easy checks on if a value is entirely missing and allows for simply representations of optional values. In the absence of a value, it will return `None` on the F# side. Null coalescing can handle the `Option` value in C# like the below example.

```
var nextInSeries = book.NextInSeries?.Value;
```

Updating a record

The most advisable way to update an record is via `non destructive mutation`. Effectively we create a new object based upon the existing one. The primary benefit to this is we are not introducing potentially unintended side effects and it remains easy to track when changes are occuring. On top of that, we maintain a version of each state of the record.

C# implementation

```
    public Book UpdateNextInSeries(Book originalBook, string nextInSeries)
    {
        return originalBook with {NextInSeries = nextInSeries};
    }
```

In this example, I have simply updated a single field in the book but I haven't destroyed or modified the original. Our returned record is a shallow copy of the previous version of the book.

F# implementation

```
let updateNextInSeries(originalBook: BookV2)(nextInSeries: string): BookV2
=
    {
        originalBook with NextInSeries = Some(nextInSeries)
    }
```

The two parameters in this case are put in two separate pairs of brackets and the return type is after the colon. But as mentioned previously, the syntax is surprisingly similar to C# 9.0 variant with the omission of braces after the `with` and treating the string as an `Option`.

Validation

Validation can be handled in many ways in the dotnet world however the cleanest ways to achieve it often requires custom validation of some kind. Constructor level checks are one of the potential approaches. For the example, I've simply utilised a mapping class.

Basic Validation

C# example

```
        if (string.IsNullOrWhiteSpace(isbn))
        {
            throw new ArgumentException("Isbn must be populated");
        }
```

The above is pretty much the most basic way of handling strings. It throws if it's not populated. Checks of this kind are required on each of these properties.

F# example

```
let validateString (candidate: string) : unit =
        if String.IsNullOrWhiteSpace candidate
        then invalidArg(nameof candidate) ""
```

In this case, I have introduced a basic null or white space check. It'll throw in the event of the value being null or whitespace.

```
          Author = ValidationModule.validateString author
          Name = ValidationModule.validateString name
```

At the mapping level the function can be called to verify required strings are populated.

Mapping

The ISBN scenario requires some more checks. Eg it should only be 10 or 13 characters long.

C# example

```
      if (string.IsNullOrWhiteSpace(isbn))
      {
          throw new ArgumentException("Isbn must be populated");
      }
      var strippedIsbn = isbn.Replace("-", "");
      return isbn.Length == 13 || isbn.Length == 10;
```

The null or whitespace checks are omitted but the logic is pretty self explanatory.

F# example

```
  let validateIsbn(isbn: string): string =
      validateString isbn |> ignore
      match (String.length (isbn.Replace("-", "")) ) with
      | 13 | 10 -> isbn
      | _ -> invalidArg(nameof isbn) ""
```

So in this case we take advantage of a match expression which allows for pattern matching. So we throw in any scenario where it doesn't match 10 or 13. It is a bit lengthier but it allows for far more complex branching of the conditions in a clear way and in a scenario where we want to deal with a particular result in the same way, it's simply a matter of adding another `|` followed by the number. We can also treat different patterns differently if we want to.

Another example of the use of it is for the page count and verifying the page count is greater than zero. We can use a match for a more concise and clear representation of the validation in F#.

F# example

```
let (|GreaterThanZero|_|) a = if a > 0 then Some() else None
let validatePageCount(pageCount: int): unit =
    match pageCount with
                    | GreaterThanZero -> ()
                    | _ -> invalidArg(nameof pageCount) "Must be greater
than 0 pages"
```

As this all comes together, we end up with the following mapper.

```
let mapBook(name: string, author: string, isbn: string,
            pageCount: int, publisher: string,
            genres: List<GenreV2>, rating: Option<int>, nextInSeries:
Option<string>): BookV2  =
    ValidationModule.validateString author
    ValidationModule.validateString name
    ValidationModule.validateIsbn isbn
    ValidationModule.validatePageCount pageCount
    ValidationModule.validateString publisher
    {
        Author = author
        Name = name
        Isbn = isbn
        PageCount = pageCount
        Publisher = publisher
        Genre = genres
        NextInSeries = nextInSeries
        Rating = rating
    }
```

It's much more simple and self explanatory than what we'd end up with in C#, validation is handled at the start of the mapper and then we can simply assign the values.

Business logic

Business logic is prone to being messy, complex conditional logic and methods that grow as issues arise are standard in industry standard code.

C# Example

For example if we want a method to search for books that are associated with a writer and genre. We can put together something like the below LINQ expression. It is somewhat functional and is not very large.

```
    public IEnumerable<Book> FindBooksByGenreAndAuthor(List<Book> books,
        string author, Genre genre)
    {
        return books.Where(x => x.Author.Equals(author)
                                        && x.Genre.Equals(genre));
        }
```

However if we wanted to allow for searching for genre or author individually, we have to decompose this method into smaller pieces.

```
        public IEnumerable<Book> FindBooksByAuthor(List<Book> books,
            string author)
        {
            return books.Where(x => x.Author == author);
        }

        public IEnumerable<Book> FindBooksByGenre(List<Book> books,
            Genre genre)
        {
            return books.Where(x => Equals(x.Genre, genre));
        }

        public IEnumerable<Book> FindBooksByGenreAndAuthorV2(List<Book> books,
            string author, Genre genre)
        {
            var booksByAuthor = FindBooksByAuthor(books, author).ToList();
            return FindBooksByGenre(booksByAuthor, genre);
        }
```

The resultant code is not awful to deal with however it is becoming a much lengthier piece of code just to achieve something very simple.

F# Example

The F# equivalent manages to be far more expressive and concise. Firstly, an engineers approach in F# tends to benefit from decomposition by default, it is easier to read when you break functionality down into smaller pieces. The syntax of the below is like a simplified version of LINQ. Each filter amounts to about 2 lines of code.

```
    let findBookByAuthor(books: List<BookV2>)(author: string): List<BookV2> =
         List.filter(fun x -> x.Author.Equals(author)) books

    let findBookByGenre (genre: GenreV2) (books: List<BookV2>): List<BookV2> =
         List.filter(fun x -> List.contains(genre) x.Genre) books

    // partial application
    let findBookByGenreAndAuthor(books:List<BookV2>) (author:string)
    (genre:GenreV2): List<BookV2> =
        findBookByAuthor books author |> findBookByGenre genre
```

In the case of `findBookByGenreAndAuthor`, we utilise `|>`(aka pipe forward). This introduced chained method calls and passes the result of `findBookByAuthor` into `findBookByGenre`. This sort of chaining would allow us to apply multiple sets of filters and passing the result down each time. On top of that you'll noticed that we don't specify an argument for books in `findBookByGenre`. This is thanks to `partial application`, thanks the ordering of our arguments, it has inferred that the piped result is our argument for the list of books. It's not even necessary to explicitly return the result as it is assumed the last expression is the expected result.

Pitfalls on both sides

C# Gone Wrong

```
        var blah = new Book
        {
        };
```

If the constructor of Book is parameterless, nothing prevents us from creating an entirely blank record on the fly. Meanwhile the F# code simply won't build if not all values are provided. The key advantage here is, F# implicitly defines the expectation of all fields being explicitly populated. We can catch common errors like the above at build. Eg forgetting a field when you have a dozen or more to define simply isn't allowed. This maintains rules around the domain by default and allows for type inference.

Other examples include the simple fact that most C# projects do not use `records` either because of lack of awareness or that projects do not support it. Some projects might do something like the below and require explicit constructors. But this often can result in pretty large and messy classes.

```
    public string Publisher { get; private set; }
```

Invariably this will lead to random properties omitting the private setter out of lazy coding. Meanwhile in F#, we default to immutable properties. Instead of explicitly defining a property as immutable, the developer must state it is `mutable`. In code review, this is far more apparent than omitting a private settter so even if it is required, a discussion can start around it.

```
    mutable Rating: Option<int>
```

F# Gotchas

Immutability?

One issue that I have encountered as part of migrating existing code to F# is the `[<CliMutable>]` attribute. For example with dapper you need a parameterless constructor. The `CliMutable` attribute effectively does that. In F#, these records will behave in the exact same way and they will remain immutable. However what it effectively does in the background is it creates getters and setters result in mutability being introduced when it gets to the C#.

```
        var publisherV2 = new PublisherV2()
        {

        };
        publisherV2.Founder = "2";
```

One scenario where this can pose a problem is when we are handling serialization with Dapper which requires a parameterless constructor for sql queries. It seems to be a somewhat common issue with a lot of libraries that handle serialization internally. Dapper will be used as the use case in this scenario as it is where issues surfaced for us.

Four potential approaches I have considered for managing this are as follows.

1. Maintain a DTO model and a separate one to return to C#.
2. Use named constructors however this is pretty time consuming
3. Accept the loss of immutability in the likes of C#
4. Handling our serialization with .Net's serialization or Newtonsoft's will allow for model's to be deserialized into an object in an immutable way.

   ```
        var result = await connection.QueryAsync<object>
   (commandDefinition);
        var serializeObject = JsonConvert.SerializeObject(result);
        return
            JsonConvert.DeserializeObject<IEnumerable<ProductV2>>
   (serializeObject,
                new OptionsConverter.OptionConverter());
   ```

Performance wise, this appears to be okay, the below results are from the Benchmark dotnet with a result set of 170 unique dapper items.When we compare the dapper serialization to the newtonsoft, the difference in performance looks to be negligible.(Serialization itself for newtonsoft was around 7 milliseconds) This converter is needed to handle options in Newtonsoft.

Method	**Mean**	**Error**	**StdDev**
RunCliMutablePureDapperVersion	437.6 ms	7.06 ms	6.93 ms
RunNewtonsoftVersion	479.6 ms	9.24 ms	21.95 ms

A sample of the extension methods for mapping is included below. This is in C# and allows for F# to remain immutable rather than becoming mutable once the object arrives in C#.

```
    public static async Task<IEnumerable<T>> QueryAsListMappedAsync<T>
(this MySqlConnection connection,
        CommandDefinition commandDefinition)
    {
        // Retrieve sql data as objects and serialize it.
        var result = await connection.QueryAsync<object>
(commandDefinition);
        var serializeObject = JsonConvert.SerializeObject(result);
        // Then deserialize it to prefered object with newtonsoft.
        return JsonConvert.DeserializeObject<IEnumerable<T>>
(serializeObject,
            new OptionConverter()) ?? new List<T>();
    }

    public static async Task<T?> QueryFirstOrDefaultMappedAsync<T>(this
MySqlConnection connection,
        CommandDefinition commandDefinition)
    {
        var result = await connection.QueryFirstOrDefaultAsync<object>
(commandDefinition);
        var serializeObject = JsonConvert.SerializeObject(result);
        return JsonConvert.DeserializeObject<T>(serializeObject,
            new OptionConverter());
    }
```

Learning curve and adoption

A different mindset needs to be applied to F#. It is possible to write F# in a way that is almost like C#. Eg our earlier filter function for books could easily be written like the below. There is nothing hugely wrong with it however it's automatically more complex to test as the individuals pieces are combined and it loses that clear aspect of piping one result to the next, instead we are falling back on variables and it is not unusual to reference the wrong one in code, the larger the function gets. On top of that, it is entirely unnecessary to explicitly state that we are returning `bookCollection`.

```
let mutable bookCollection = []

let findBookByGenreAndAuthor(books:  List<BookV2>) (author:string)
(genre:GenreV2): List<BookV2> =
    let filteredAuthors = List.filter(fun x -> x.Author.Equals(author))
books
    bookCollection <- List.filter(fun x -> List.contains(genre) x.Genre)
filteredAuthors
    bookCollection
```

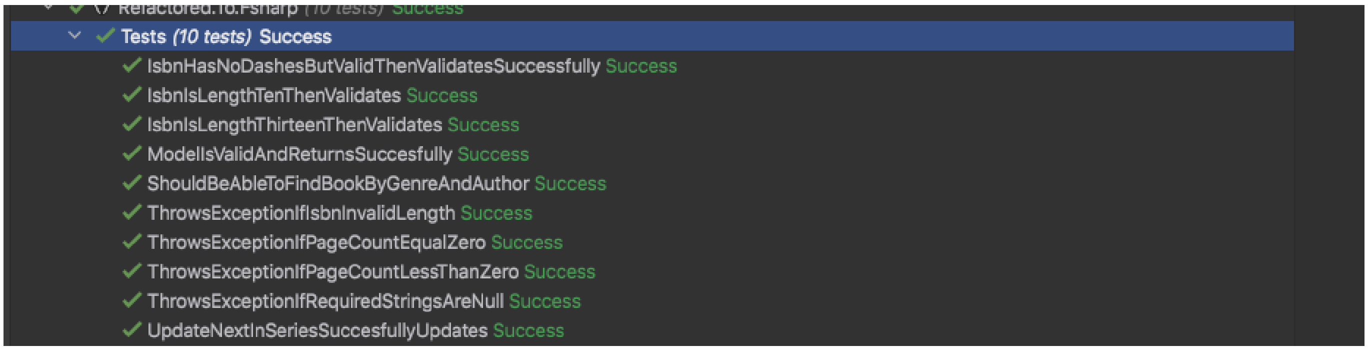
However the most insulting part of this code is impurity. This is a somewhat stupid example but we have enabled the mutation of the `bookCollection`. The back pipe `<-` has been utilised to mutate it. However we have made the function `impure` and have potentially introduced impurity into other functions that rely on that collection. In the original version, the result of the function was always predictable, however the out of scope `bookCollection` has introduced potential side effects.

A lot of issues such as the above are common sense issues and can easily be avoided by not enabling things like mutation. `Purity` is a key concept to functional programming in general, if a function receives the same argument, the result should always be predictable. Some parts of moving one's thought process from object oriented to functional concepts are a hurdle. It is possible to write F# like C# however much of time things like for loops or if-else conditions are simply unnecessary.

So initially, refactoring existing code over to F# will be slower for those transitioning from object oriented design.

Unit Tests

Some basic unit testing was introduced to verify the functionality. The testing approach itself is pretty easy since everything is decomposed.



```
  ✓ <> Refactored.To.Fsharp (10 tests) Success
    ✓ Tests (10 tests) Success
      ✓ IsbnHasNoDashesButValidThenValidatesSuccessfully Success
      ✓ IsbnIsLengthTenThenValidates Success
      ✓ IsbnIsLengthThirteenThenValidates Success
      ✓ ModelIsValidAndReturnsSuccesfully Success
      ✓ ShouldBeAbleToFindBookByGenreAndAuthor Success
      ✓ ThrowsExceptionIfIsbnInvalidLength Success
      ✓ ThrowsExceptionIfPageCountEqualZero Success
      ✓ ThrowsExceptionIfPageCountLessThanZero Success
      ✓ ThrowsExceptionIfRequiredStringsAreNull Success
      ✓ UpdateNextInSeriesSuccesfullyUpdates Success
```

Conclusion

This article has only really touched upon issues and approaches towards refactoring C# to F#. However it should hopefully give an idea of the immense benefits that F# offers to existing dotnet projects. Everything from LINQ to C# 9.0's records borrow heavily from functional programming in general so the huge benefit that F# offers is you are also able to offer such functionality as immutability to legacy projects that don't necessarily support C# 9.0.

Interoperability can admittedly be unpredictable at times. Eg the immutability issue with CliMutable. However as demonstrated, it is straightforward to achieve it with some experimentation. With immutability, we gain the advntage of pure code. Unexpected side effects such as our end results varying are bypassed. Plus it's incredibly concise and can easily be integrated into existing C# code. The biggest hurdle is transitioning from object oriented thinking to functional. When closely abiding by functional principles, it is possible to write code that is clear and self explanatory.