Shipping Pseudocode to Production

VarnaConf 2019

“Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer’s intent but rather is full of crisp abstractions and straightforward lines of control.”

- **Grady Booch** - author of “*Object Oriented Analysis and Design with Applications*”

```
RowData ParseRow(ExcelRow row);
```

```
ExcelRow[] rows = ReadExcelRows();

RowData[] parsedData = rows
    .Select(ParseRow)
    .ToArray();
```

```
try
{
    ExcelRow[] rows = ReadExcelRows(excelFile);

    RowData[] parsedData = rows
        .Select(ParseRow)
        .ToArray();
}
catch (ParsingException e)
{
    // Handle
}
```

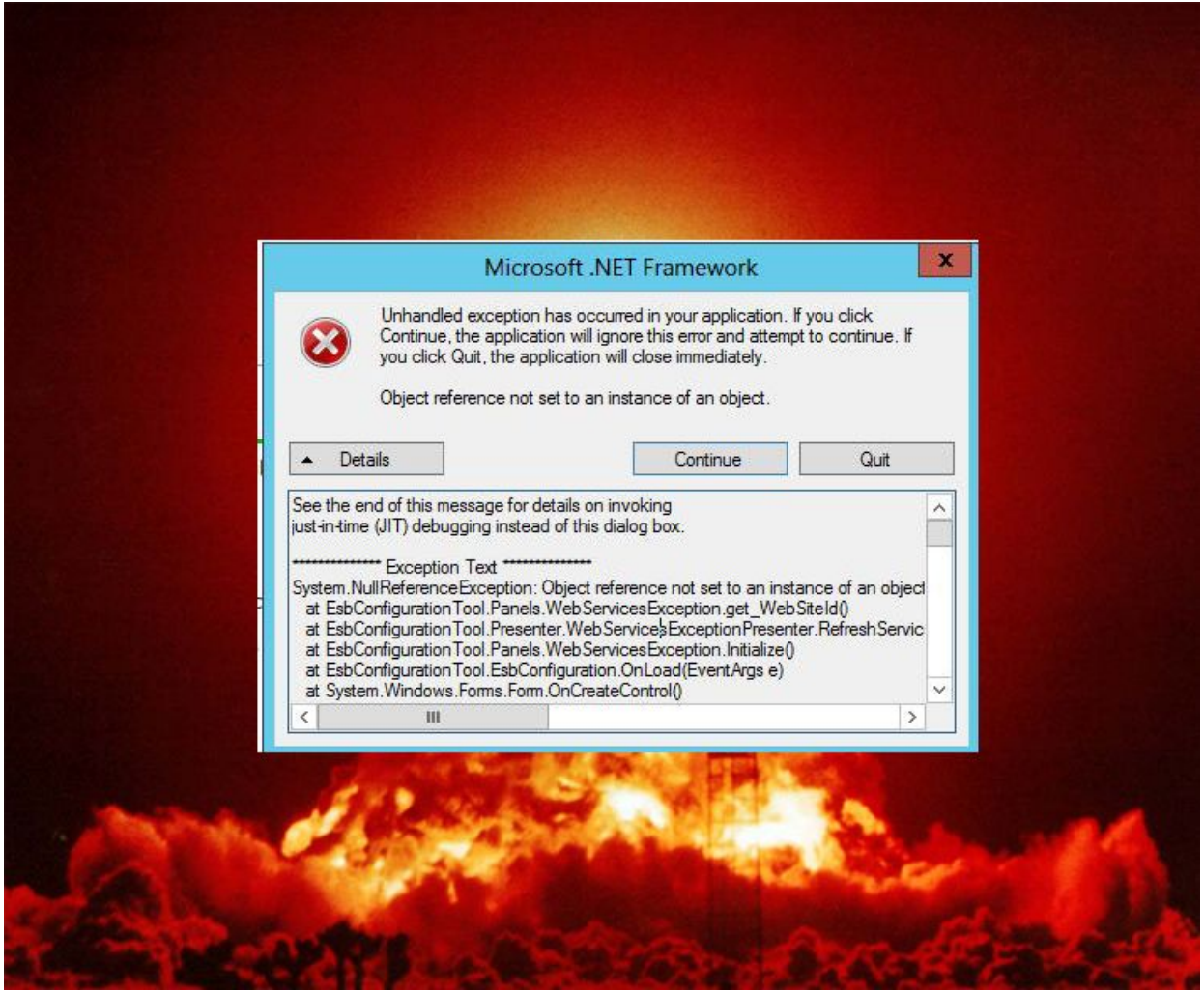

Geez! Do we seriously pay these guys to code?

```
class RowData
{
    ...
    string Error { get; set; }
    ...
}
```

```
class ParsingResult
{
    RowData Data { get; set; }
    string Error { get; set; }
}
```

```
ParsingResult ParseRow(ExcelRow row);
```

```
ParsingResult[] parsedData = rows
    .Select(ParseRow)
    .ToArray();

string[] errors = parsedData
    .Select(r => r.Error)
    .ToArray();
```


Microsoft .NET Framework
Unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.
Object reference not set to an instance of an object.
Details
Continue
Quit
See the end of this message for details on invoking just-in-time (JIT) debugging instead of this dialog box.
********** Exception Text **********
System.NullReferenceException: Object reference not set to an instance of an object
at EsbConfigurationTool.Panels.WebServicesException.get_WebSiteId()
at EsbConfigurationTool.Presenter.WebServicesExceptionPresenter.RefreshServic
at EsbConfigurationTool.Panels.WebServicesException.Initialize()
at EsbConfigurationTool.EsbConfiguration.OnLoad(EventArgs e)
at System.Windows.Forms.Form.OnCreateControl()

```
ExcelRow row = ReadExcelRow();
ParsingResult result = ParseRow(row);

if (result.Error == null)
{
    // Do something with result.Data
}
else
{
    // Handle the error
}
```

If anything can be forgotten, it will be forgotten.

Why?

Expressiveness

Expressiveness through types

Our first expressive function

Performing HTTP requests

```
async Task<Uri> CreateProductAsync(Product product)
{
    HttpResponseMessage response = await client.PostAsJsonAsync(
        "api/products", product);
    response.EnsureSuccessStatusCode();

    // return URI of the created resource.
    return response.Headers.Location;
}
```

```
async Task<Uri> CreateProductAsync(Product product)
{
    HttpResponseMessage response = await client.PostAsJsonAsync(
        "api/products", product);
    response.EnsureSuccessStatusCode();

    // return URI of the created resource.
    return response.Headers.Location;
}
```

Our new HTTP request function

- HTTP has 62 status codes but **only 10 of them mean success**
- **84%** of the status codes mean some kind of error

therefore

- The caller should be signaled unambiguously that errors are likely to occur
- It should be easy for the caller to handle the potential errors

How?

- Through its name (*PerformAnHttpRequestAndRememberItsLikelyToFail*)
- Through its arguments (*successHandler* and *errorHandler*)
- Through its return type

- Through its name (*PerformAnHttpRequestAndRememberItsLikelyToFail*) ✔️
- ~~Through its arguments (*successHandler* and *errorHandler*)~~
- ~~Through its return type~~

- ~~Through its name (*PerformAnHttpRequestAndRememberItsLikelyToFail*)~~
- Through its arguments (*successHandler* and *errorHandler*)
- Through its return type

- ~~Through its name (*PerformAnHttpRequestAndRememberItsLikelyToFail*)~~
- ~~Through its arguments (*successHandler* and *errorHandler*)~~
- Through its return type

- ~~Through its name (*PerformAnHttpRequestAndRememberItsLikelyToFail*)~~
- ~~Through its arguments (*successHandler* and *errorHandler*)~~
- Through its return type ✔️

Designing our type

Requirements

1. It should be able to contain 2 other types - one that is going to be returned if the operation went successfully and another one if it failed
    a. In the HTTP request case, we would like one type of response if the server returned a success status code and another one if it failed
2. It should never be null
    a. It makes no sense for a type that signals a likely error to be null as that will lead to even more errors

```
public struct Either<T, TException>
{
   // Constructors omitted for brevity
   public bool IsSuccessful { get; }

   public static Either<T, TException> Success(T value) =>
       new Either<T, TException>(value);

   public static Either<T, TException> Error(TException exception) =>
       new Either<T, TException>(exception);
}
```

```
Either<SucessfulHttpResponse, HttpError> PerformHttpRequest(...)
```

```
Either<SucessfulHttpResponse, HttpError> responseResult =
    PerformHttpRequest("GET", "http://someurl.com/");

// ??????
```

```
public struct Either<T, TException>
{
   public bool IsSuccessful { get; private set; }

   public void Match(Action<T> success, Action<TException> error)
   {
       if (IsSuccessful)
           success(value);
       else
           error(exception);
   }

   public TResult Match<TResult>(Func<T, TResult> success, Func<TException, TResult> error) =>
       IsSuccessful ? success(value) : error(exception);
}
```

```
Either<SucessfulHttpResponse, HttpError> responseResult =
    PerformHttpRequest("GET", "http://someurl.com/");

// Compiler error
responseResult.Match(
    some: response => ...
)
```

```
Either<SucessfulHttpResponse, HttpError> responseResult =
    PerformHttpRequest("GET", "http://someurl.com/");

responseResult.Match(
    some: response => /* Happy path */,
    none: error => /* Sad path */
);
```

Performing consecutive requests

```
HttpResponse response = HttpRequest(...);

if (response.IsSuccessfull)
{
    HttpResponse anotherResponse = HttpRequest(... response.Data);

    if (anotherResponse.IsSuccessfull)
    {
        HttpResponse thirdResponse = HttpRequest(... anotherResponse.Data);

        if (thirdResponse.IsSuccessfull)
        {

        }
    }
}
```

```
var responseResult = HttpRequest(...);

responseResult.Match(
    some: response =>
    {
        var anotherResponseResult = HttpRequest(... response.Data);

        anotherResponseResult.Match(
            some: anotherResponse =>
            {
                var thirdResponseResult = HttpRequest(... anotherResponse.Data);

                thirdResponseResult.Match(
                    some: thirdResponse
                    {

                    },
                    error: /* Error handling */
                )
            },
            error: /* Error handling */
        )
    },
    error: /* Error handling */
)
```

```
// Perform a request
// If the request is successful, perform another request
// If the consequent request is successful, perform another request
// ...
```

```
// Perform a request
// If the request is successful, perform another request
// If the consequent request is successful, perform another request
// If the consequent request is successful, perform another request
```

```
// Perform a request

// If the request operation is successful, perform another request operation

// If the consequent request operation is successful, perform another request
operation

// If the consequent request operation is successful, perform another request
operation
```

```
public struct Either<T, TException>
{
    public Either<TResult, TException> Map<TResult>(Func<T, TResult> mapping) =>
        Match(
            some: value => Either.Success<TResult, TException>(mapping(value)),
            none: exception => Either.Error<TResult, TException>(exception)
        );
}
```

Vadete bonbonite

```
public struct Obvivka<TBonbona>
{
    private readonly TBonbona _bonbona;

    public static Obvivka<TResult> Map(Func<TBonbona, TResult> operaciq) =>
        Obvivka.SlojiVytre(operaciq(_bonbona));
}
```

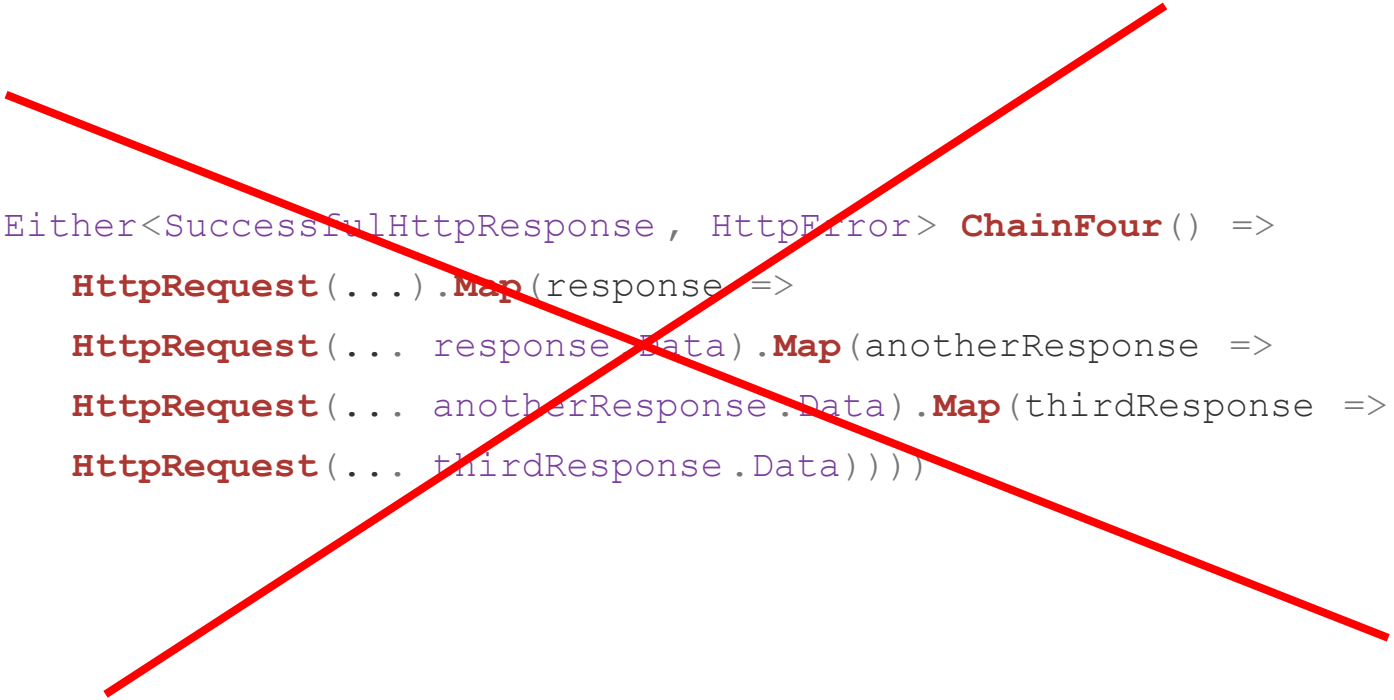
```
OblizanaBonbona Obliji(Bonbona bonbona);
```

```
Bonbona bonbonaLukche = new BonbonaLukche();

Obvivka<Bonbona> lukche = Obvivka.SlojiVytre(bonbonaLukche);

Obvivka<OblizanaBonbona> lukcheSOblizanaBonbona = lukche.Map(Obliji);
```

```
public struct Either<T, TException>
{
    public Either<TResult, TException> Map<TResult>(Func<T, TResult> mapping) =>
        Match(
            some: value => Either<TResult, TException>.Success(mapping(value)),
            none: exception => Either<TResult, TException>.Error(exception)
        );
}
```

```
var responseResult = HttpRequest(...);

responseResult.Map(response =>
{
    var anotherResponseResult = HttpRequest(... response.Data);

    anotherResponseResult.Map(anotherResponse =>
    {
        var thirdResponseResult = HttpRequest(... anotherResponse.Data);

        thirdResponseResult.Map(thirdResponse =>
        {
        })
    })
})
```

```
HttpRequest(...).Map(response =>
HttpRequest(... response.Data).Map(anotherResponse =>
HttpRequest(... anotherResponse.Data).Map(thirdResponse =>
HttpRequest(... thirdResponse.Data))))
```



```
Either<SuccessfulHttpResponse, HttpError> ChainFour() =>
    HttpRequest(...).Map(response =>
    HttpRequest(... response.Data).Map(anotherResponse =>
    HttpRequest(... anotherResponse.Data).Map(thirdResponse =>
    HttpRequest(... thirdResponse.Data))))
```

The type of

```
HttpRequest(...).Map(response =>
HttpRequest(... response.Data).Map(anotherResponse =>
HttpRequest(... anotherResponse.Data).Map(thirdResponse =>
HttpRequest(... thirdResponse.Data))))
```

is

```
Either<Either<Either<Either<SucessfulHttpResponse, HttpError>,
HttpError>, HttpError>, HttpError>
```

Vadete bonbonite

```
Obvivka<OblizanaBonbona> OblijiIZavii(Bonbona bonbona);
```

```
Bonbona bonbonaLukche = new BonbonaLukche();

Obvivka<Bonbona> lukche = Obvivka.SlojiVytre(bonbonaLukche);

Obvivka<Obvivka<OblizanaBonbona>> dvoinoZavitoOblizanoLukche =
    lukche.Map(OblijiIZavii);
```

```
// Either<SuccessfulHttpResponse, HttpError>
HttpRequest(...).Map(response =>
// Either<Either<SuccessfulHttpResponse, HttpError>, HttpError>
HttpRequest(... response.Data).Map(anotherResponse =>
// Either<Either<Either....>
HttpRequest(... anotherResponse.Data).Map(thirdResponse =>
// Either<Either<Either<Either...>
HttpRequest(... thirdResponse.Data))));
```

```
Either<Either<Either<Either<SucessfulHttpResponse , HttpError>, HttpError>,
HttpError>, HttpError>

->

Either<SucessfulHttpResponse , HttpError>
```

Vadete bonbonite

```
public struct Obvivka<TBonbona>
{
    private readonly TBonbona _bonbona;

    public static Obvivka<TResult> Map(Func<TBonbona, TResult> operaciq) =>
        Obvivka.SlojiVytre(operaciq(_bonbona));
}
```

```
public struct Obvivka<TBonbona>
{
    private readonly TBonbona _bonbona;

    public static Obvivka<TResult> FlatMap(Func<TBonbona, Obvivka<TResult>> operaciq) =>
        operaciq(_bonbona);
}
```

```
Bonbona bonbonaLukche = new BonbonaLukche();

Obvivka<Bonbona> lukche = Obvivka.SlojiVytre(bonbonaLukche);

Obvivka<OblizanaBonbona> dvoinoZavitoOblizanoLukche =
   lukche.FlatMap(OblijiIZavii);
```

```
public struct Either<T, TException>
{
   public Either<TResult, TException> FlatMap<TResult>(Func<T,
Either<TResult, TException>> mapping) =>
       Match(
           some: mapping, // We skip wrapping the value into an Either
           none: exception => Either.Error<TResult, TException>(exception)
       );
}
```

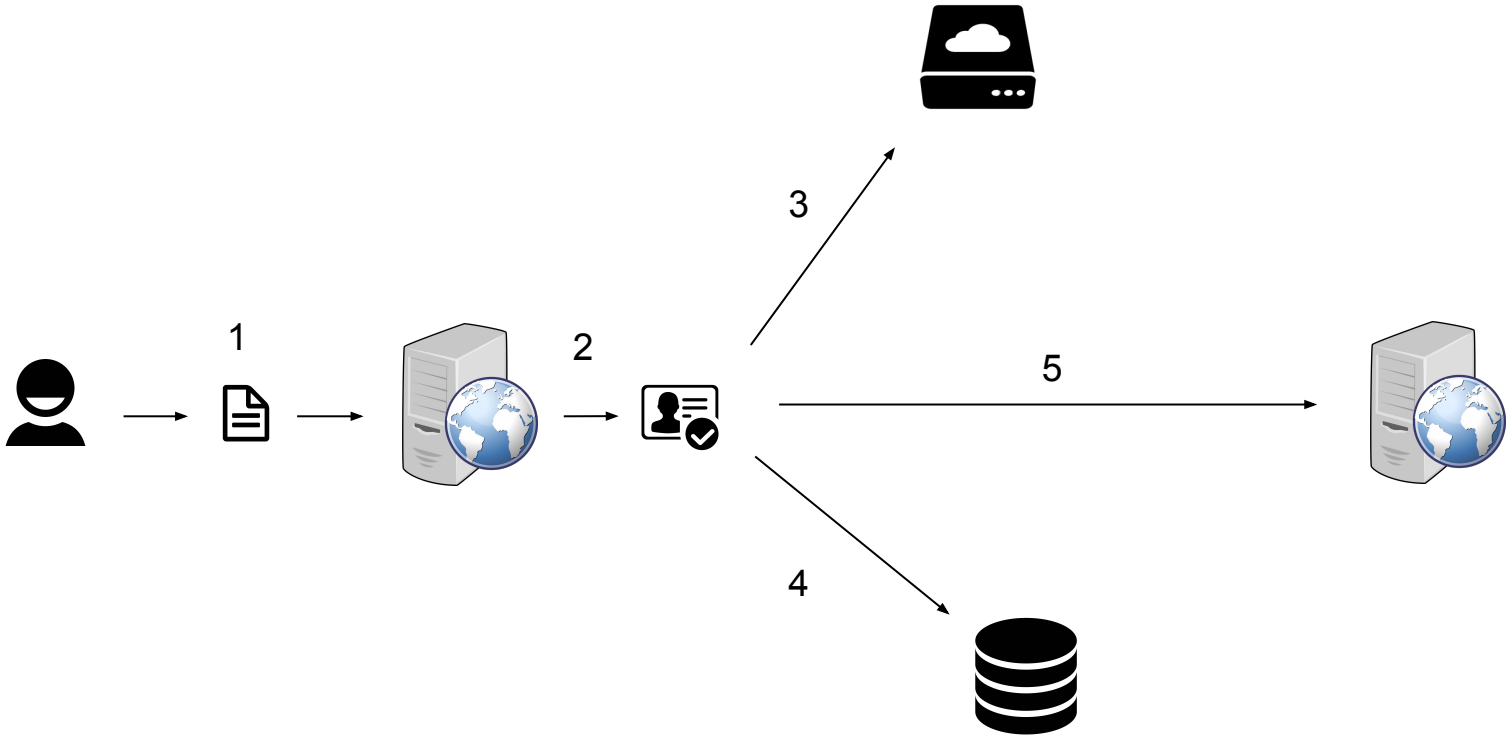
```
HttpRequest(...).FlatMap(response =>
HttpRequest(... response.Data).FlatMap(anotherResponse =>
HttpRequest(... anotherResponse.Data).FlatMap(thirdResponse =>
HttpRequest(... thirdResponse.Data))))
```

```
Either<SuccessfulHttpResponse, HttpError> ChainFour() =>
    HttpRequest(...).FlatMap(response =>
    HttpRequest(... response.Data).FlatMap(anotherResponse =>
    HttpRequest(... anotherResponse.Data).FlatMap(thirdResponse =>
    HttpRequest(... thirdResponse.Data))))
```

```
HttpResponse response = HttpRequest(...);

if (response.IsSuccessfull)
{
    HttpResponse anotherResponse = HttpRequest(... response.Data);

    if (anotherResponse.IsSuccessfull)
    {
        HttpResponse thirdResponse = HttpRequest(... anotherResponse.Data);

        if (thirdResponse.IsSuccessfull)
        {
            HttpResponse fourthResponse = HttpRequest(... thirdResponse.Data);

            if (fourthResponse.IsSuccessfull)
            {

            }
        }
    }
}
```

```
HttpRequest(...).FlatMap(response =>
HttpRequest(... response.Data).FlatMap(anotherResponse =>
HttpRequest(... anotherResponse.Data).FlatMap(thirdResponse =>
HttpRequest(... thirdResponse.Data))))
```

Going into production


1
2
3
4
5

```
// function ProcessDocument userId, category, file
//   1. check if the user is authorized
//   2. store the file into the cloud
//   3. store database log with unique key
//   4. send the unique key to an external service for post-processing
```

```
Either<User, Error> CheckIfUserIsAuthorized(string userId, string category);

Either<CloudRecord, Error> StoreTheFileIntoTheCloud(File file, string
category);

Either<Guid, Error> StoreDatabaseLog(CloudRecord record);

Either<DocumentProcessedResult , Error> SendToExternalService(Guid key);
```

```
Either<DocumentProcessedResult , Error> ProcessDocument(
    string userId,
    string category,
    File file) =>
    CheckIfUserIsAuthorized(userId, category).FlatMap(user =>
    StoreTheFileIntoTheCloud(file, category).FlatMap(cloudRecord =>
    StoreDatabaseLog(cloudRecord).FlatMap(uniqueKey =>
    SendToExternalService(uniqueKey))));
```

“Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer’s intent but rather is full of crisp abstractions and straightforward lines of control.”

- **Grady Booch** - author of “*Object Oriented Analysis and Design with Applications*”

1. Simple and direct - ✔️
2. Reads like well-written(?) prose - ✔️
3. Doesn’t obscure the designer’s intent - ✔️
4. Full of crisp abstractions and straightforward lines of control - ✔️

```
DocumentProcessedResult ProcessDocument(
  string userId,
  string category,
  File file)
{
  var user = _userRepository.GetById(userId);
  if (user != null)
  {
      var isAuthorized = _authService.IsUserAuthorized(user, category);
      if (isAuthorized)
      {
          try
          {
              var cloudRecord = _cloudStorageService.Store(file, category);
              var databaseStorageResult = _recordsRepository.Save(new DatabaseRecord
              {
                  Key = Guid.NewGuid(),
                  CloudRecordId = cloudRecord.Id
              });
              // ...and so on
          }
          catch (CloudStorageException)
          {
              // Handle
          }
          catch (DatabaseException)
          {
              // Handle
          }
      }
  }
}
```

Where next?

- *“Shipping Pseudocode to Production”* article (link)
- A sample application using DDD and event-sourcing with complete integration tests coverage (link)
- A CLI tool for managing your music collection (link)
- Real life examples of Either in C# (link)

Q&A

Thanks!