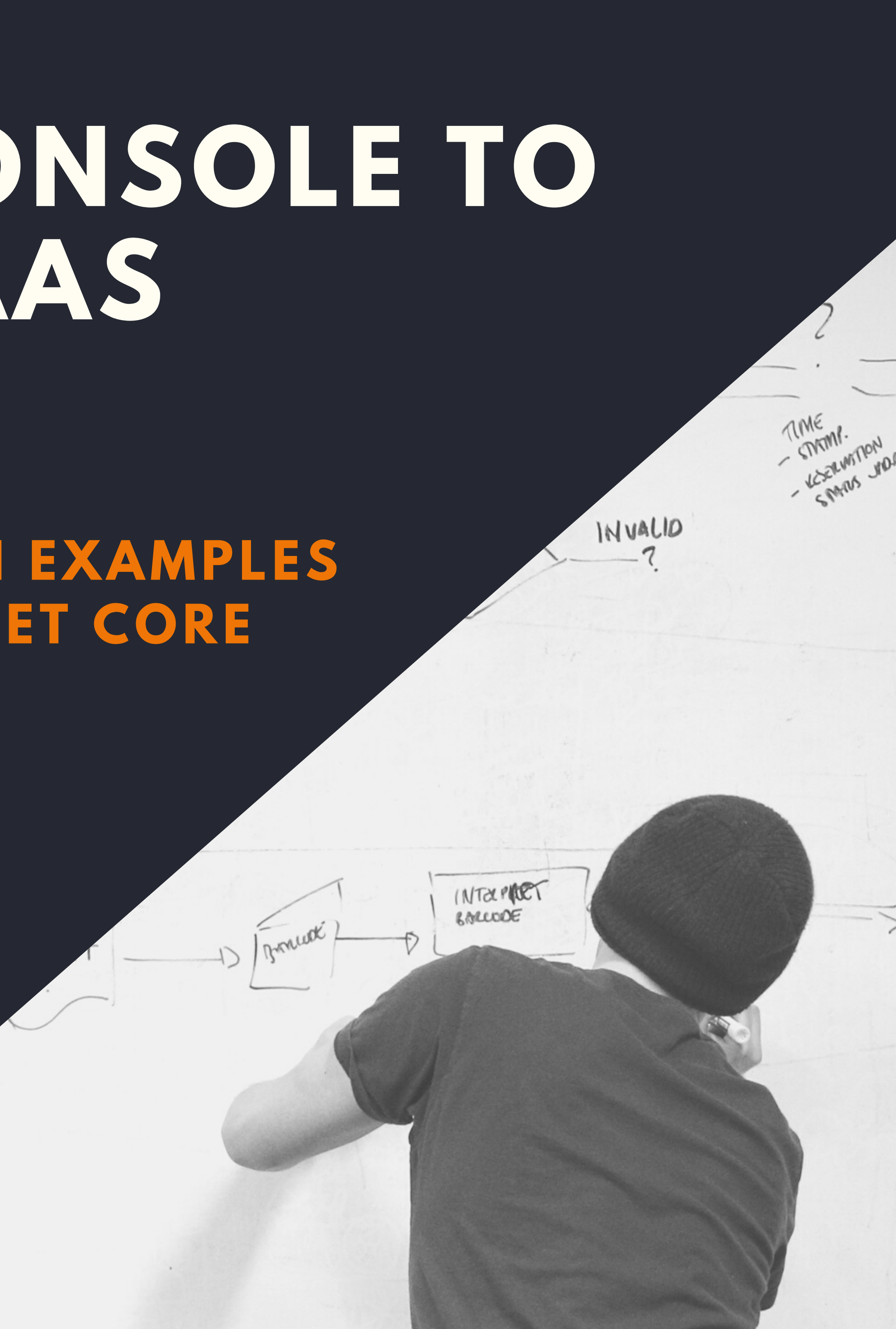
CONSOLE TO SAAS

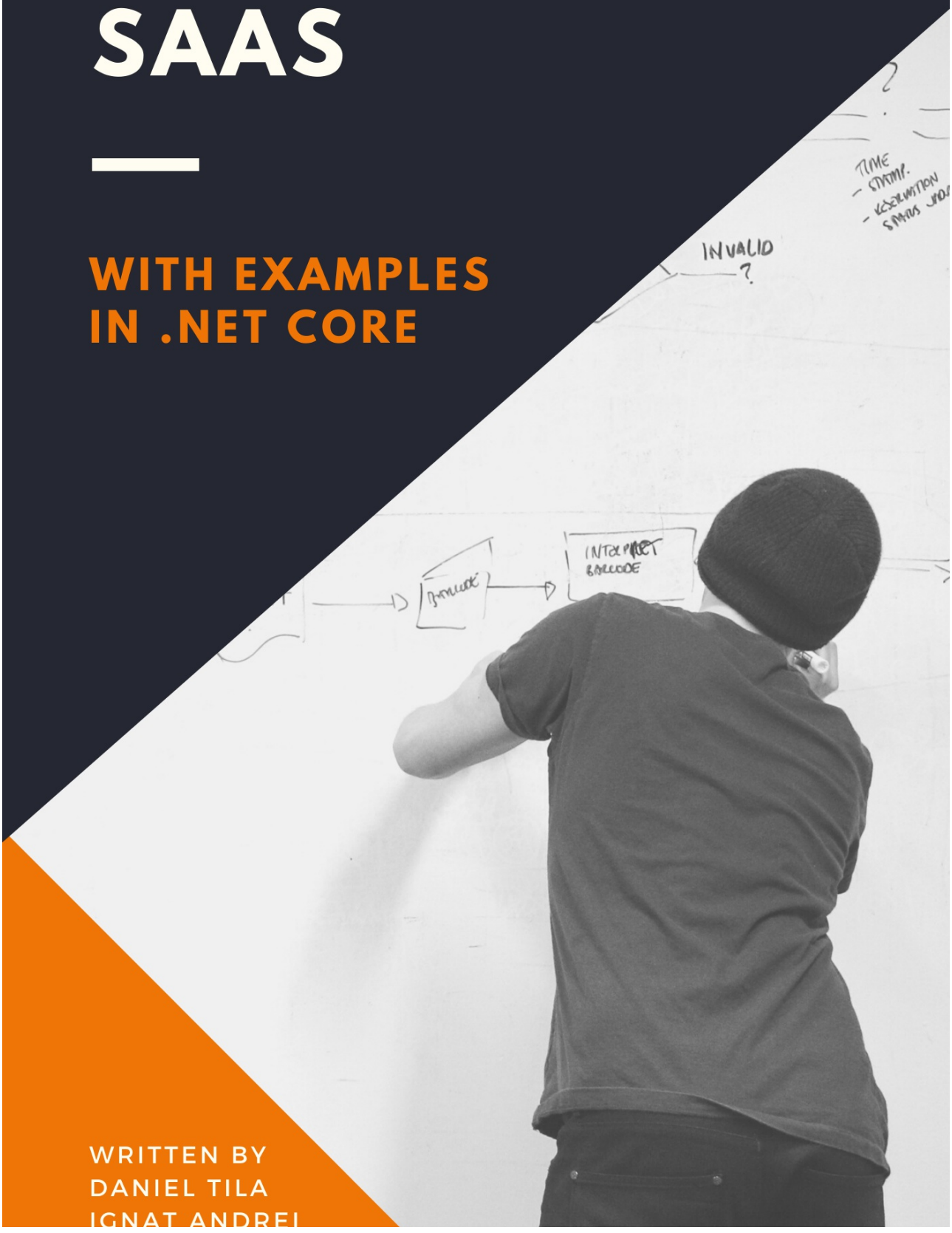
WITH EXAMPLES IN .NET CORE



WRITTEN BY
DANIEL TILA
IGNAT ANDREI

CONSOLE TO SAAS

WITH EXAMPLES IN .NET CORE



WRITTEN BY
DANIEL TILA
IGNAT ANDREI

Console to SAAS

How to transform a Proof Of Concept application to a Software As A Service product

This book will guide you step-by-step how you can build a scalable product from a proof of concept (https://en.wikipedia.org/wiki/Proof_of_concept) to a production ready SAAS (https://en.wikipedia.org/wiki/Software_as_a_service).
Any development done will start from a business need: this will make things clear for the team what is the impact of the delivery.
You will see different arhitecture patterns for separation of concerns (https://en.wikipedia.org/wiki/Separation_of_concerns) and why some of them fit well and some of them not. Everything will happen incrementally and the product development history will be easy to be seen by analyzing commits.
To have our finished product, we will see the two development directions:

On premise solution

- how the idea get a shape
- creating the PoC
- bug fixing
- componentization / testing / refactoring
- unit and integration testing
- extending functionalities

Scaling

- authentication
- load testing
- profiling and optimisations
- continuous deployment
- tight coupled microservices
- messaging systems / loose coupled microservices
- third party integrations

Download the book for free

You can find the code source at
https://github.com/ignatandrei/console_to_saas

You can read online this book at https://ignatandrei.github.io/console_to_saas/

You can read offline by downloading the

1. PDF (https://ignatandrei.github.io/console_to_saas/consoleToSaas.pdf.html)
2. Word (https://ignatandrei.github.io/console_to_saas/consoleToSaas.docx.html)
3. Epub (https://ignatandrei.github.io/console_to_saas/consoleToSaas.epub.html)

4. ODT (https://ignatandrei.github.io/console_to_saas/consoleToSaas.odt.html)

Source codes

You can find source codes at Console2SAAS (https://github.com/ignatandrei/console_to_saas/) or after each chapter.

How the chapters are organized

Each chapter has the following steps:

1. A *description* of the business challenge
2. An *exercise* for the reader
3. A *technical analysis* - what should be modified in order to do the project
4. Some *code* that shows how the challenge is solved (also, link to the full code to be downloaded)
5. *Further reading* - if the reader wants to learn more

What we will build

The application will be a simple one: from multiple word documents we will build a summarizing Excel file. The point here is not to build the application 100% production-ready, but our focus will be on code organization and discuss the common problems while on this journey.

Feedback

Please share feedback by opening an issue or pull request at https://github.com/ignatandrei/console_to_saas/

Authors

Andrei Ignat - http://msprogrammer.serviciipeweb.ro/

Has more then 20 year programming experience. He started from VB3, passed via plain old ASP and a former C# Microsoft Most Valuable Professional (MVP).
He is also a consultant, author, speaker, www.adces.ro community leader and http://www.asp.net/ moderator.

You can ask him any .NET related question – he will be glad to answer – if he knows the answer. If not, he will learn.

Daniel Tila - is a systems engineer master degree in system architecture and 10+ years of experience. I worked on airplane engine software, train security doors to bank payment and printing systems.

I am a certified teaching UiPath Developer & Architect and co-founder at Automation Pill where we help companies to optimize their costs by automating repetitive, mundane tasks. Holding training courses has always attracted me and I really enjoy a curious audience to whom I can interact with and challenge with fun practical exercises

Chapter 1

The boss that is hurrying up

True story: one day my boss came and he asked me to discuss how can we help our colleagues from the financial department since they are thinking about the automation process. We found a spare place and scheduled a meeting where we find out that 3 employees were manually extracting the client information (name, identity information, address) and contract agreement details (fee, payment schedule) to an Excel file and saving the document in a protected area.

Exercise

Make an application that reads all word documents from a folder, parses, and outputs the contents to Excel. Create and make it work, the easy way. You can find sample data in the folder data in the GitHub project.

Technical analysis

After putting some brainstorming and analyzing the constraint we ended up developing a solution that reads from a folder share and continues the process without any human intervention. This would allow us to have all our employees to continue their work simultaneously with less effort than before. To speed up the things, we decided to go with a minimal setup

- prototype solution in a console app
- use NPOI (https://github.com/dotnetcore/NPOI) library for handling Word document reading. The decision is simply based on preference. ClosedXML (https://github.com/ClosedXML/ClosedXML) is another options. However, we have chosen to go with an approach that doesn't require to have the Microsoft Office installed. Limiting the dependencies is generally a good approach and in our case, it was easier also.

We wrapped everything in the same project, just to make sure it works and to keep our focus on delivering. Everything we put in the main file of the console app, and we have chosen C# because we were the most comfortable and were able to fulfill our requirements in a minimum amount of time.

Code

```
string folderWithWordDocs = Console.ReadLine();

//omitted code for clarity
foreach (string file in Directory.GetFiles(folderWithWordDocs, "*.docx"))
{
    //omitted code for clarity
    var contractorDetails = ExactContractorDetails(document.Tables[0]);
    allContractors.Add(contractorDetails);
}
```

We were facing resolving the solution just with a list of sequences of operations without any control flow involved. Great!

Download code

Code at Chapter01 files:1;lines:66 (https://ignatandrei.github.io/console_to_saas/sources/Chapter01.zip)

Further reading

How to create a menu for the console app - https://github.com/migueldeicaza/gui.cs

When reading files on the hard disk, you should understand if you want to enumerate all files or just see if there are any files.
Read about the difference between IEnumerable and array (e.g. Directory.EnumerateFiles vs Directory.GetFiles) - https://www.codeproject.com/Articles/832189/List-vs-IEnumerable-vs-IQueryable-vs-ICollection-v

Chapter 2

Adapt to the financial department last tweaks

We installed our console app on one of the computers of an employee in the financial department, we set it up and we let it run. After a few days, we had some bug fixing and some new requests that came in. We saw an enthusiastic behavior on our colleagues by using our solution and that was our first real feedback that we were resolving a customer pain. We anticipated that we will get more feedback and to track what we have done we needed a source control. After a few iterations, we had our financial department happy and we are ready with our first MVP.

Problem

Try to refactor the solution from Chapter 1 to extract the business logic (i.e. transforming Word files into an Excel file) into a separate file.

Technical analysis

As a starting point, we decided to have 2 separate projects:

- a business logic
- the host (in our case the console app)

We decide to have this approach because it keeps a well balanced effort between value and time, and allows the flexibility to ensure separation of concerns (https://en.wikipedia.org/wiki/Separation_of_concerns).
We went with creating the **WordContractExtractor** where we handle all the Word related file formats. Here we moved the usage of the NPOI library and we make sure that the expected format is according to our requirements.
Hence, our solution looks now like:

- all the file conversion resides in a single place
- the main program file is more simple

This is a simple and long term solution because it is not a time-waster and offers us the possibility to adapt later to possible changes. Moreover using this technique early in the process, helps to isolate the problem better. You have less code to look in one place, and the problems are isolated.

To help the versioning we added a source control. We have chosen GIT since it is simple and allows us to work without an internet connection.

Code

```
var wordExtractor = new WordContractExtractor(folderWithWordDocs);
wordExtractor.ExtractToFile(excelResultsFile);
```

Now, all we need to do is to use the class and pass the needed parameters. In this case, the file path is the only dependent requirement. This path can come from a config file or from a user pick action. The only check we could add is to make sure it is writeable. We can add this check before

executing the program or we can just handle separately the exception and display to the user a message. We will see that later in the next chapters.

Download code

Code at Chapter02 files:2;lines:86 (https://ignatandrei.github.io/console_to_saas/sources/Chapter02.zip)

Further reading

What is version control: https://en.wikipedia.org/wiki/Version_control

Create a free version control repository at http://github.com/, https://azure.microsoft.com/en-us/services/devops/, or other online source control.

Read more about how you improve the design of existing code by refactoring: https://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672

Chapter 3

Second internal customer

Sharing code source + settings for client

It has been a few months since our financial department is using the application and they were happy with it. On a morning coffee break where we were discussing the new automation trends, they mention the tool that we developed. The sales department was willing to see since they had themselves some similar situation.
We configured and deployed on a separate machine for them and we saw great feedback and real usage on a daily basis. They were so excited to see how they decreased the paperwork hours and had more time to pay attention to the client's needs.
They want to use the application as an example of automation and we find it more suitable as product development.

Problem

Make a Console solution that can read a configurable setting (the folder path to be searched) from a file instead of being hardcoded into the application. Do not over-engineer the solution. Create and make it work, the easy way.

Technical analysis

Creating a configuration file with all the settings that a client can particularly change. For now, we have just the directory path, along with the config. We have added also a class that manages the reading operation to move the reading operation

The reason that we put the setting into a configuration file and not make 2 applications is to have an easy way to maintain the same application for 2 clients. Also, later, the client may want to read from 2 different folders?

The impact on the solution is minimal. Just create a new class that knows how to read from a config file and retrieve the name of the folder as a property.

Code

```
var settings = Settings.From("app.json");
var directoryToSearch= settings.DocumentSLocation;
```

Download code
Code at Chapter03 files:3;lines:117
(https://ignatandrei.github.io/console_to_saas/sources/Chapter03.zip)

Further reading

How an application can have multiple configuration sources (environment, command line, configuration file, code - in this order) - Read about .NETCore endpoint configuration (https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel?view=aspnetcore-2.2)

Hot reload the configuration: read IOptionSnapshot and IOptionMonitor at https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options?view=aspnetcore-3.1

Chapter 4

Transform to product

Making a Graphical User Interface

At our Digital Transformation Center Department, we have recurring meetings with clients that are part of our continuous iterative process to identify things that need to be improved. In one of the meetups that we hosted on our desk, we invited the financial department to leave feedback on the process that we helped. We talked about how the process evolved and how by adopting Digital Solutions decreased manual work.
The presentation ended up with some Q&A and several clients they were interested to have the application installed. At that point, we realized that this is a missing point in the current market and we planned to develop a product around this. At this point, we had a validated idea by the market and we decided to evolve based on the client's needs.
Selling the application to multiple clients means to have configurable the word extracting and location of the documents. We ended up creating a GUI to allow the user to configure himself with the automation parameters. This was a valuable feature for our users since most of them could be non-technical.
Transfer the ownership of the paths from the config file to a more appropriate user interface.

Problem

Modify the existing solution to support both Console and GUI.

Technical analysis

For both applications, we have the same functionality: reading files from the hard disk and reading settings. We do not want to copy-paste code and become un-maintainable, so we want to share code source from a console application to the GUI application.

Because the user works now with network folders that cannot be available always, we need to perform some modifications to alert the user for problems. In our case we have the user interface for showing :

- the progress of the operations
- Adding logging for knowing what happens if an error occurs
- Adding exception handling to not crash the application
- Handle edge cases of the user saving configuration (assert that user have not entered any folder / maybe list what documents have been successfully processed and whatnot)

From a solution with a single project, we have now a solution with 3 projects:

- Business Layer (BL) - the logic for the application - in our case, processing documents from a folder
- Console (referencing BL)
- Desktop (referencing BL)

The BL has the code that was previously in the console application (loading settings, doing extraction) + some new code (logging, others).

Code

The business logic is now more simple:

```
public void ExtractToFile(string excelFileOutput)
{
string[] files = Directory.GetFiles(_documentLocation, "*.docx");
_logger.Info($"processing {files.Length} word documents");
//code omitted for brevity
```

The Console code now is very simple, just calling code from BL:

```
var settings = Settings.From("app.json");
var extractor = new WordContractExtractor(settings.DocumentsLocation);
extractor.Start();
```

The Desktop is calling the same BL, with some increased feedback for the user:

```
try
{
    string folderWithWordDocs = folderPath.Text;
    if (string.IsNullOrWhiteSpace(folderWithWordDocs))
    {
        MessageBox.Show("please choose a folder");
        return;
    }
    string excelResultsFile = "Contractors.xlsx";
    var wordExtractor = new WordContractExtractor(folderWithWordDocs);
    wordExtractor.ExtractToFile(excelResultsFile);
}
catch(Exception ex)
{
    _logger.Error(ex,$"exception in  {nameof(StartButton_Click)}");
    MessageBox.Show("an error occured. See the log file for details");
}
```

Download code

Code at Chapter04 files:6;lines:287
(https://ignatandrei.github.io/console_to_saas/sources/Chapter04.zip)

Further reading

.NET Core 3.0 Windows application (WPF, WinForms): https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-core-3-0

MultiTier architecture: https://en.wikipedia.org/wiki/Multitier_architecture

Chapter 5

The first real client

Componentization / Testing / Refactoring

A client who was interested mentioned that they were using zip files for backup the documents for company service. They were interested in having our product, but we didn't support zip files. We shared our vision to create a highly customizable product, so implementing reading files from zipping would be an additional feature. He agreed to pay for it and we started the product development.

Problem

Starting from the solution, add support for reading either from a folder, either from a zip file.

Technical analysis

Changing the business core requires refactoring which usually involves regression testing. Before every refactoring is started, we need to define the supported test scenarios from the business perspective. This is a list that usually includes inputs, appropriate action, and the expected output. The easiest way to achieve this is to add unit tests which do the checking automatically.

Altering pretty much the entire core adds some regression risks. To compensate for that, we will add tests (unit and/or integration and/or component and/or system) which verifies the already old behavior and the new one. The tests will make sure that our core (reading and parsing) outputs the expected output using various file systems. We will add a new project for unit/component test and we will reference the business core project and inject different components (in our case, file system).
This core modification will also impact other projects, including GUI and console applications.
In our case, we want to make sure that the contract parsing success if we use both file systems.

We already identified the operations that need to be abstract, so let's write the interfaces that allow this:

1. An interface for the file system which supports listing files (IFileSystem)
2. An interface with the actual file. This needs to have a Name and to read the content from either a zip file, either a folder into the system (IFile).

Our core program needs to:

1. find Word documents
2. extract information from it

These two operations are part of the actual business flow. We need to abstract the finding operation and reading from a file operation and allow different implementations.

Code

One implementation is from the actual file system (an existing directory) and the other one from the zip archive, hence:

1. DirectoryFileSystem
2. ZipFileSystem

A zip file can have inside folders and files, similar to a directory. Hence, we can do the same operations just like in the actual file system (listing files and reading a file from it).

```
var settings = Settings.From("app.json");
var fileSystem = settings.FileSystemProvider.CurrentFileSystem();
var extractor = new WordContractExtractor(fileSystem);
extractor.Start();
```

Download code

Code at Chapter05 files:12;lines:475 (https://ignatandrei.github.io/console_to_saas/sources/Chapter05.zip)

Homework

There are a few issues that you could encounter to transform this code example in production. The WordContractExtractor depends on the IFileSystem. The latter one is the state (it keeps the source files). While it is ok to have a state which is read-only (multiple requests will not alter the state), you could have concurrency problems if the files are written there. In this case, the IFileSystem has only methods for reading. Your homework is to modify the WordContractExtractor to be consistent and use the IFileSystem for writing.

Your task is to:

- add a function for writing in the IFileSystem that receives a Stream and it writes the content to the file system.
- add a property CanWrite to the IFileSystem and ensure writing is possible
- implement in the classes that IFileSystems inherits from the functionality

Further reading

1. Choosing Between an Interface and an Abstract Class - https://medium.com/better-programming/choosing-between-interface-and-abstract-class-7a078551b914
2. How to manage external connections using IDisposable - https://docs.microsoft.com/en-us/dotnet/api/system.idisposable
3. Read about file system abstraction (exists already: https://github.com/System-IO-Abstractions/System.IO.Abstractions
4. How to unit test http://dontcodetired.com/blog/post/Unit-Testing-C-File-Access-Code-with-SystemIOAbstractions)