Design Pattern & Clean Code

Refactoring PDFTools to Patterns

Thomas Ley | @CleanCodeCoach

Goals

- Refactoring of PdfTools project
- Branch Cleaner-Code with refactorings
- Each refactoring has a dedicated commit
- Each commit has a descriptive commit message
- Each commit has one or more descriptive slides
- The focus lies on the pattern, and keeps irrelevant parts crappy

Implement [Command Pattern]

The goal of this refactoring is moving the old "if-else-statements" from `Program.cs` into dedicated classes.
Each class has only one responsibility [SRP] to provide the functionality for a dedicated action (command).

Implement [Command Pattern]

- [GitHub Commit](#)

- Add `ICommand` interface as abstraction

- Move "if-else" code to command implementation per action

- Register available commands with their action name and make code more generic

Enhanced command pattern

The goal of this enhancement is using the power of reflection and C# option to make the code more flexible and extensible in a generic way.

Attributes are used to create the available commands dictionary based on all implemented commands.

Enhanced command pattern

- GitHub Commit

- Add `CommandNameAttribute` to each command implementation

- Implement `CommandHelper` to get a list of all commands and their action/command name

- Enhance `ICommand` to provide a usage help text

- Create `HelpCommand` to print user help

- Replace exceptions with help messages in `Progam.cs`

[Zero Impact Injection] QR code

The goal of this refactoring is extracting the QR code creation code to dedicated class and use this class instead of copy-paste code in each class.

In addition, the extracted class gets an interface to be more flexible according the implementation and testability.

[Zero Impact Injection] QR code

- GitHub Commit

- Identify similar code

- Move code to method (for QR code already done)

- Move method to class

- Extract interface

- Inject class through interface into using class

- Use default implementation to avoid breakting changes (no command changed!)

Unit Tests

The goal of this refactoring is implementing basic testing. It will create a test project and create a sample test.

The test project uses NUnit and FluentAssertions (library to make test assertions more readable).

Basically there are two strategies with advantages and disad

Unit Tests

- [GitHub Commit](#)
- Create dedicated test project
- Add sample test for previously created QR code generator: `QrCoderServiceTest`
- I like `Be_Creatable()` test to check constrcutor and constrains like interfaces
- Test happy path for `CreateOverlayImage(string)`
- Test error path for `CreateOverlayImage(null)`
- A null text returns an empty QR code

Separate PDF Handler

The goal of this refactoring is separating concerns more clearly and handle download in a single class instead of copy-paste code in both classes.

The separation leads to a single class with separated methods.

Separate PDF Handler

- [GitHub Commit](#)

- Create a `PdfHandler` class

- Separate the concerns of `PdfArchiver` and `PdfCodeEnhancer` methods

- Make smaller but generic methods to handle the request

- Refactor the commands to use the new class.

- Review Code and check for additional places to change

Clean-up Principles

The following clean-up code principles help to make the code better, if every team member acts this was.

- **Broken Windows Principle**: When you start adding crappy code, time by thime, everyone will follow and create crappy code, too.

- **Boyscout Rule**: Leave a place in a better shape than you found it. Rename variables to match the purpose better, refactor small code parts etc.

- **30 Second Rule**: If it only takes 30 seconds, do it now and do not put it onto the bench.

Cleanup Principles

- [GitHub Commit](#)

- Refactor variable names to get rid of the legacy names

- Identify crappy code and simply use a previously refactored class

- Clean namespaces regularily and remove unused variables

[Static Class Wrapper]

The goal of this refactoring is the testability of the pdf handler regarding regarding the http download.

An interface and wrapper is created for HttpClient so it can be injected through an interface. In this example, all `Http*` return values are abstracted and wrapped with an interface and only the used methds are added.

After that, "Zero Impact Injection" is used to have non breaking changes and default behaviour. `Commands` have not changed.

[Static Class Wrapper]

- [GitHub Commit](#)

- Add interface `IHttpClient` and inject interface to class

- Use interface in class (so we can get the used methods easily)

- Create a wrapper `HttpClientWrapper` and implement interface.

- Add a "static class wrapper" for `HttpResponseMessage` and `HttpContent`

- In opposite to `HttpClientWrapper`, provide the wrapped instance as constructor parameter

[Dispose Pattern]

The goal of this refactoring is implementing the (C#/.NET) dispose pattern so we can clean the resources we used by our code.

In our `PdfHandler` we can delete the temporary file which is created and modified by the class.

See external Documentation for more Information on Dispose/Finalize.

[Dispose Pattern]

- [GitHub Commit](#)

- Interface and implementation added to wrapper so they dispose their 'wrapee'

- IDisposable pattern in a clean form in `PdfHandler`

- Delete temporary file if `Dispose()` is called

Use `System.Io.Abstractions`

The goal of this refactoring is using an existing library for [Static Class Wrapper] on `System.IO` namespace. The library is called `System.Io.Abstractions` and uses the same pattern, we used before.

Use `System.Io.Abstractions`

- [GitHub Commit](#)
- Remove "[System.IO](#)" namespace everywhere in `PdfHandler` classes
- Inject the interface "IFileSystem" and use it for file access
- Use default implementation `FileSystem` (which is a factory for FileSystem*Wrapper)

Clean Up the code

After all the refactoring to a single class or only a subset of classes, some code cleanups are still outstanding.

This refactring will clean up the missing classes and components and just apply the previous chapters to all parts of the code.

Clean Up the code

- [GitHub Commit](#)

- Move all pdf handling and transformation code to `PdfHandler`

[Factory Pattern]

The goal of this refactoring is separating the creation and the handling of pdf documents.

A factory will be created to create a pdf handler instance with an initial document. After that, the processing is done independent from creation.

[Factory Pattern]

- [GitHub Commit](#)

- Create factory class `PdfHandlerFactory` to create and return a handler `PdfHandler`

- Use interfaces for the factory `IDocumentHandlerFactory` and the handler `IDocumentHandler`

- Inject factory into commands and use `PdfHandlerFactory` as default

Dispose Pattern for testing

This refactoring uses the dispose pattern from .NET to inject or override some parts and restore it after the test is done.

The implementation of "IDisposable" injects an imlementation in `constructor` and resets it during `dispose`.

If you find code or solutions on the internet (e.g. stackoverflow), put a link to our code so other know, what you did.

Dispose Pattern for testing

- [GitHub Commit](#)

- Add class in test project which implements IDisposable (`ConsoleOutputForTest`)

- Inject custon code to console using `Console.Setout` and cache old value

- Reset injection in dispose and restore "normal" behaviour

- Use class through
  `using (var consoleOutput = new ConsoleOutputForTest())` in `ProgramTest`

- Use a `// cf:` comment to point to the internet where you find

Summary

Fact: The complexity of the code and interfaces has increased.

Different concerns have interfaces for testability and have a lot more classes and methods than just the extracted code.

But on the opposite, the code is more flexible (e.g. strategy pattern) and easier to extend (e.g. command pattern).

Open Tasks

The follwing slides contain upcoming refactorings and code improvements which are not yet done.

TODO: [Service Locator]

The goal of this refactoring is implementing a service locator to move all `new()` to a dedicated class.

The class will be a proof of concept with dedicated `GetServiceA()` methods and a generic `GetService<T>`.

TODO: [Service Locator]

- [GitHub Commit](#)
- Create a `ServiceLocator` class to create
  - Commands
  - Factory
  - All Dependencies

TODO: Testing the Code

TODO: Move this to another presentation?

The goal of this refactoring is testing some of the code which was refactored in the last commits. This shows all the advantages of our refactorign and explains gained testability.

TODO: Testing the Code

- GitHub Commit
- Create a test class for each class
- Create a 'fake' or 'mock' for each dependency (NSubstitute)
- Inject dependency behaviour to know how the dependency behaves

TODO: Future Refactorings

- Refactor commands to inject dependencies and use a service locator to create the object graph

- Decorate the downloader with a QR code overlay adder

- Facade in HttpClient wrapper so we can get rid of two interfaces.