# Design Pattern & Clean Code

## What are Dependencies?

Thomas Ley | @CleanCodeCoach

# Indirection

> "All problems in computer science can be solved by another level of indirection"

**Butler Lampson**

in ***"fundamental theorem of software engineering"***

# Dependencies

- Class (internal, hard)
- Interface (internal, soft)
- Libraries (external, hard/soft).

# Class

☝ **New is Glue**

- "Hard dependency"
- Knows implementation
- Cannot be replaced
- `new()` as indicator.

# Interface

- "Soft dependency"
- Hides implementation
- Can be replaced
- Subset of functionality [ISP].

# Libraries

- "Hard dependency" or "Soft dependency"
- Knows or hides implementation
- Can partially be replaced
- Depending from 3rd party.

# Independence is Freedom

- Reduce (or remove) hard dependencies
- Use abstractions (e.g. Interfaces, Lambda)
- Watch your dependencies
- Define your dependencies.

# Dependencies can be found

- nuget/references
- namespace usings
- constructor
- `new()`
- properties.

# Favour Composition

- Favour composition over inheritance
- Legacy Clean Code principle
- Multi-Inheritance (C++)
- A class using inheriance to "get functions".

# Design Pattern & Clean Code

## Refactor `PdfTools`

Thomas Ley | @CleanCodeCoach

# Refactoring `PdfTools`

- Written without any design
- Some pdf and qrcode features
- Download of files
- Concatenate files.

# Issues in `PdfTools`

- Duplicate code
- Un-testable
- No [SRP]
- Hard to extend
- Hard to reuse components.

# Demo "PdfTools"

# Challenge

- How do I clean up brownfield projects?
- Without rewriting them?
- Without breaking them?
- Where do I start?

# Design Pattern & Clean Code

## Zero Impact Injection

Thomas Ley | @CleanCodeCoach

# Goals

- Refactor Brownfield Project
- Single Responsibility Principle
- Interface Segregation Principle
- Dependency Injection
- Zero Impact Injection
- Testable Code.

# Steps

- Extract Code (Method ➡️ Class ➡️ Project)
- Create Single Dependency ( `new()` )
- Inject Dependency
- Null-Object Pattern.

# Extract code [SRP]

- Extract method
- Move to class
- Introduce field

💡 Namespace indicates dependencies
💡 Create a documentation of the class without "and"
🆚 **Demo: Barcode Code**

# [Zero Impact Injection]

- Add interface to class
- Inject as interface into class
- Use `??` for default implementation

**VS Demo: Barcode class**

# [Null-Object Pattern]

- Reduces null-checks
- "Identity Element" for an interface
- Empty implementation
- Returns `default`
- IFooBar --> EmptyFooBar
- Implemented along with Interface

🆚 **Demo: Empty logger class**

# Design Pattern & Clean Code

## Dependency Injection

Thomas Ley | @CleanCodeCoach

# [DIP]

❗ The Dependency Inversion Principle (DIP) states that high level modules should not depend on low level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions. ❗

# Goals

- DI & SL differences
- Lifetime scopes
- Factory Pattern

# DI vs. SL

- [Dependency injection] (DI)
  - Injects dependencies
  - Contructor/property injection
- [Service Locator] (SL)
  - Single point of contact
  - Static dependency resolver.

# Demo

- Project EUMEL Dj
- Mobile app uses service locator
- Desktop app uses dependency injection.

# DI as SL

- Inject DI container
- Resolve dependency from container.

# Lifetime Scopes

- Unique-Instance context or scope
- Example
  - Per process
  - Per thread
  - Per HTTP request
  - Any customer defined
- DI framework has already implementations.

# [Singleton]

- Instance is created once
- Instance is created on first use
- [Double-Check Locking]
- MSDN Documentation
- Implementation see Lazy<T>.

# Demo

- Project src/Zapfenstreich.sln

# AD: `DI Frameworks`

- Reduces "hard dependencies"
- Delegates creation
- Simplifies injecting of code
- Simplifies changing of implementation

☝ A DI container makes you write cleaner software
👍 A DI container helps refactoring code.

# Factories

- Creates an *implemenentation*
- Returns an *interface*.

# [Factory] Implementations

- Class with `Create()` method(s)
- Interface with `Create()` method(s)
- Func<T> / Lambda.

# Lazy<T> vs. Func<T>

- Func<T> is a method which creates a T
- Lazy<T> implements a singleton
- Lazy gets a Func as constructor parameter
- Lazy can solve circular (DI) dependencies.

# Demo

- Project

# Design Pattern & Clean Code

## Testable Code

Thomas Ley | @CleanCodeCoach

# Goals

- Pattern for "testable code"
- Separate "untestable code".

# Unit tests

- Cost time and money
- Changes when code changed
- No value for the customer
- Learn an additional concept.

# Unit tests

- Not only check input --> output
- They define what happens in error-execution path
  - Logging
  - Return values
  - Rethrow exception?
- The document the usage of a class
  - Parameter usage
  - Return values
  - Expected behaviour

# Unit tests

- Double check code
- Detect unwanted changes
- Keeps your code clean.

# About testing

- Myth: One assert per test
- Truth: One path per test.

# About testing

- Myth: Each method has a test method
- Truth: Each method will have multiple test methods
  - One "happy path"
  - Multiple "error path".

# Testing

- Brings all pieces together
- Clean code to test [SRP] classes
- [DIP] to mock dependencies.

# [Static Class Wrapper]

- Create Wrapper
- Extract interface

☝ Used for Static classes
☝ Used for classes without interface

# Mock HttpClient

- Create HttpClientWrapper
- Extract interface IHttpClient
- Create HttpClientMock
- Use interface and inject implementation

# Demo

- Project

# Random(), DateTime()

- Same pattern applies to "changing" data
- Predict random numbers
- Change date time during test (e.g. test cache expiration)

# "Mocking" Frameworks

- NSubstitute
- System.Io.Abstractions
- ContextFor<T>

# Demo

- Project

# Architectural Tests

- Unit tests to verify architectural decisions
- E.g. Each class must have a test
- Operator methods must be virtual

# Design Pattern & Clean Code

## Pattern, Clean Code and Best Practices

Thomas Ley | @CleanCodeCoach

# Goals

- Design Pattern
- Clean Code
- Best Practices

# Kaizen vs. Kaikaku

- `Make the world a better place, day by day`
- Kaizen is the Sino-Japanese word for "improvement"
- Kaikaku is the Japanese term for "radical change"

# [Boyscout Rule]

```
Always leave the code better than you found it.
```

- Remove technical debts
- Renaming, documentation, tests, or
- Clean(er) Code
- Small or smaller, depends on you

# [Broken Window Principle}

- If you start breaking it,
- Everyone will continue.

# [30 second rule]

`If it takes only five minutes, do it now`

- If it can be fast, do it now
- If it is a bigger task, create a ticket
- If there are too many "five mins", create a `TODO`

# SOLID

- Single-responsibility principle
  - Every class should have only one responsibility
- Open–closed principle
  - Software entities … should be open for extension, but closed for modification.
- Liskov substitution principle
  - A derived class should not break behaviour of the base class
- Interface segregation principle
  - Many client-specific interfaces are better than one general-purpose interface

# Extension Methods

- No state
- No business logic
- Just "facade" methods

# Aggregation Pattern

- IGetInformation
- IAggregateGetInformation
- IAggregateGetInformation implements IGetInformation
- IAggregateGetInformation registers only with aggregate interface

# Tagging Interface

- Empty interface to register a class for a specific purpose

# 3rd party libraries

- Create separate project
- Interfaces to "contracts" project
- Empty implementation to "contracts" project
- Concrete implementation in 3rd party library

# [Strategy Pattern]

- Inject behaviour
- Strategy changes depending on needs
- DI implementa strategy with registrations
- Some frameworks support "named registrations"

💡 Demo: StrategyResolver<T>

# [Command Pattern]

- Is a strategy pattern
- Has the following methods
  - `CanExecute(context)`
  - `Execute(context)`

💡 Demo: PdfTools

# 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

# 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.