Design Patterns in .NET Core 3

Reusable Approaches in C# and F# for Object-Oriented Software Design

—

Second Edition

—

Dmitri Nesteruk

Apress®

Design Patterns in .NET Core 3

Reusable Approaches in C# and F# for Object-Oriented Software Design

Second Edition

Dmitri Nesteruk

Apress®

Design Patterns in .NET Core 3: Reusable Approaches in C# and F# for Object-Oriented Software Design

Dmitri Nesteruk
St. Petersburg, c.St-Petersburg, Russia

ISBN-13 (pbk): 978-1-4842-6179-8 ISBN-13 (electronic): 978-1-4842-6180-4
https://doi.org/10.1007/978-1-4842-6180-4

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Joan Murray
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484261798. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Dmitri Nesteruk is a quantitative analyst, developer, course and book author, and an occasional conference speaker. His interests lie in software development and integration practices in the areas of computation, quantitative finance, and algorithmic trading. His technological interests include C# and C++ programming as well as high-performance computing using technologies such as CUDA and FPGAs. He has been a C# MVP from 2009 to 2018.

About the Technical Reviewer

Adam Gladstone has over 20 years of experience in investment banking, building software mostly in C++ and C#. For the last couple of years, he has been developing data science and machine learning skills, particularly in Python and R after completing a degree in Maths and Statistics. He currently works at Virtu Financial in Madrid as an Analyst Programmer. In his free time, he develops tools for NLP.

Introduction

The topic of Design Patterns sounds dry, academically dull, and, in all honesty, done to death in almost every programming language imaginable – including programming languages such as JavaScript which aren't even properly object-oriented! So why another book on it? I know that if you're reading this, you probably have a limited amount of time to decide whether this book is worth the investment.

I decided to write this book to fill a gap left by the lack of in-depth patterns books in the .NET space. Plenty of books have been written over the years, but few have attempted to research all the ways in which modern C# and F# language features can be used to implement design patterns, and to present corresponding examples. Having just completed a similar body of work for C++,[1] I thought it fitting to replicate the process with .NET.

Now, on to design patterns – the original *Design Patterns* book[2] was published with examples in C++ and Smalltalk, and since then, plenty of programming languages have incorporated certain design patterns directly into the language. For example, C# directly incorporated the Observer pattern with its built-in support for events (and the corresponding `event` keyword).

Design Patterns are also a fun investigation of how a problem can be solved in many different ways, with varying degrees of technical sophistication and different sorts of trade-offs. Some patterns are more or less essential and unavoidable, whereas other patterns are more of a scientific curiosity (but nevertheless will be discussed in this book, since I'm a completionist).

Readers should be aware that comprehensive solutions to certain problems often result in overengineering, or the creation of structures and mechanisms that are far more complicated than is necessary for most typical scenarios. Although overengineering is a lot of fun (hey, you get to *fully* solve the problem and impress your co-workers), it's often not feasible due to time/cost/complexity constraints.

[1] Dmitri Nesteruk, *Design Patterns in Modern C++* (New York, NY: Apress, 2017).

[2] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading, MA: Addison Wesley, 1994).

Who This Book Is For

This book is designed to be a modern-day update to the classic Gang of Four (GoF, named after the four authors of the original Design Patterns book) book, targeting specifically the C# and F# programming languages. My focus is primarily on C# and the object-oriented paradigm, but I thought it fair to extend the book in order to cover some aspects of functional programming and the F# programming language.

The goal of this book is to investigate how we can apply the latest versions of C# and F# to the implementation of classic design patterns. At the same time, it's also an attempt to flesh out any new patterns and approaches that could be useful to .NET developers.

Finally, in some places, this book is quite simply a technology demo for C# and F#, showcasing how some of the latest features (e.g., default interface methods) make difficult problems a lot easier to solve.

On Code Examples

The examples in this book are all suitable for putting into production, but a few simplifications have been made in order to aid readability:

- I use public fields. This is not a coding recommendation, but rather an attempt to save you time. In the real world, more thought should be given to proper encapsulation, and in most cases, you probably want to use properties instead.
- I often allow too much mutability either by not using `readonly` or by exposing structures in such a way that their modification can cause threading concerns. We cover concurrency issues for a few select patterns, but I haven't focused on each one individually.
- I don't do any sort of parameter validation or exception handling, again to save some space. Some very clever validation can be done using C# 8 pattern matching, but this doesn't have much to do with design patterns.

You should be aware that most of the examples leverage the latest version of C# and generally use the latest C# language features that are available to developers. For example, I use `dynamic`, pattern matching, and expression-bodied members liberally.

At certain points in time, I will be referencing other programming languages such as C++ or Kotlin. It's sometimes interesting to note how designers of other languages have implemented a particular feature. C# is no stranger to borrowing generally available ideas from other languages, so I will mention those when we come to them.

Preface to the Second Edition

As I write this book, the streets outside are almost empty. Shops are closed, cars are parked, public transport is rare and empty too. Life is almost at a standstill as the country endures its first "nonworking month," a curious occurrence that one (hopefully) only encounters once in a lifetime. The reason for this is, of course, the COVID-19 pandemic that will go down in the history books. We use the phrase "stop the world" a lot when talking about the garbage collector, but this pandemic is a *real* "stop the world" event.

Of course, it's not the first. In fact, there's a pattern there too: a virus emerges, we pay little heed until it's spreading around the globe. Its exact nature is different in time, but the mechanisms for dealing with it remain the same: we try to stop it from spreading and look for a cure. Only this time around it seems to have really caught us off guard, and now the whole world is suffering.

What's the moral of the story? Pattern recognition is critical for our survival. Just as the hunters and gatherers needed to recognize predators from prey and distinguish between edible and poisonous plants, so we learn to recognize common engineering problems – good and bad – and try to be ready for when the need arises.

PART I

Introduction

CHAPTER 1

The SOLID Design Principles

SOLID is an acronym which stands for the following design principles (and their abbreviations):

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

These principles were introduced by Robert C. Martin in the early 2000s – in fact, they are just a selection of five principles out of dozens that are expressed in Robert's books and his blog. These five particular topics permeate the discussion of patterns and software design in general, so, before we dive into design patterns (I know you're all eager), we're going to do a brief recap of what the SOLID principles are all about.

Single Responsibility Principle

Suppose you decide to keep a journal of your most intimate thoughts. The journal has a title and a number of entries. You could model it as follows:

```
public class Journal
{
  private readonly List<string> entries = new List<string>();
  // just a counter for total # of entries
  private static int count = 0;
}
```


D. Nesteruk, *Design Patterns in .NET Core 3*, https://doi.org/10.1007/978-1-4842-6180-4_1

Now, you could add functionality for adding an entry to the journal, prefixed by the entry's ordinal number in the journal. You could also have functionality for removing entries (implemented in a very crude way in the following). This is easy:

```
public void AddEntry(string text)
{
  entries.Add($"{++count}: {text}");
}

public void RemoveEntry(int index)
{
  entries.RemoveAt(index);
}
```

And the journal is now usable as

```
var j = new Journal();
j.AddEntry("I cried today.");
j.AddEntry("I ate a bug.");
```

It makes sense to have this method as part of the `Journal` class because adding a journal entry is something the journal actually needs to do. It is the journal's responsibility to keep entries, so anything related to that is fair game.

Now, suppose you decide to make the journal persist by saving it to a file. You add this code to the `Journal` class:

```
public void Save(string filename, bool overwrite = false)
{
  File.WriteAllText(filename, ToString());
}
```

This approach is problematic. The journal's responsibility is to *keep* journal entries, not to write them to disk. If you add the persistence functionality to `Journal` and similar classes, any change in the approach to persistence (say, you decide to write to the cloud instead of disk) would require lots of tiny changes in each of the affected classes.

I want to pause here and make a point: an architecture that leads you to having to do lots of tiny changes in lots of classes is generally best avoided if possible. Now, it really depends on the situation: if you're renaming a symbol that's being used in a hundred places, I'd argue that's generally OK because ReSharper, Rider, or whatever IDE you use

will actually let you perform a refactoring and have the change propagate everywhere. But when you need to completely rework an interface… well, that can become a very painful process!

We therefore state that persistence is a separate *concern*, one that is better expressed in a separate class. We use the term *Separation of Concerns* (sadly, the abbreviation SoC is already taken[1]) when talking about the general approach of splitting code into separate classes by functionality. In the cases of persistence in our example, we would externalize it like so:

```
public class PersistenceManager
{
  public void SaveToFile(Journal journal, string filename,
                         bool overwrite = false)
  {
  if (overwrite || !File.Exists(filename))
    File.WriteAllText(filename, journal.ToString());
  }
}
```

And this is precisely what we mean by *Single Responsibility*: each class has only one responsibility and therefore has only one reason to change. `Journal` would need to change only if there's something more that needs to be done with respect to in-memory storage of entries; for example, you might want each entry prefixed by a timestamp, so you would change the `Add()` method to do exactly that. On the other hand, if you wanted to change the persistence mechanic, this would be changed in `PersistenceManager`.

An extreme example of an anti-pattern[2] which violates the SRP is called a *God Object*. A God Object is a huge class that tries to handle as many concerns as possible, becoming a monolithic monstrosity that is very difficult to work with. Strictly speaking, you can take any system of any size and try to fit it into a single class, but more often than not, you'd end up with an incomprehensible mess. Luckily for us, God Objects are easy to recognize either visually or automatically (just count the number of member functions), and thanks to continuous integration and source control systems, the responsible developer can be quickly identified and adequately punished.

[1]SoC is short for System on a Chip, a kind of microprocessor that incorporates all (or most) aspects of a computer.

[2]An *anti-pattern* is a design pattern that also, unfortunately, shows up in code often enough to be recognized globally. The difference between a pattern and an anti-pattern is that anti-patterns are typically patterns of bad design, resulting in code that's difficult to understand, maintain, and refactor.

Open-Closed Principle

Suppose we have an (entirely hypothetical) range of products in a database. Each product has a color and size and is defined as follows:

```
public enum Color
{
  Red, Green, Blue
}

public enum Size
{
  Small, Medium, Large, Yuge
}

public class Product
{
  public string Name;
  public Color Color;
  public Size Size;

  public Product(string name, Color color, Size size)
  {
    // obvious things here
  }
}
```

Now, we want to provide certain filtering capabilities for a given set of products. We make a `ProductFilter` service class. To support filtering products by color, we implement it as follows:

```
public class ProductFilter
{
  public IEnumerable<Product> FilterByColor
    (IEnumerable<Product> products, Color color)
  {
    foreach (var p in products)
      if (p.Color == color)
        yield return p;
```

```
  }
}
```

Our current approach of filtering items by color is all well and good, though of course it could be greatly simplified with the use of Language Integrated Query (LINQ). So, our code goes into production but, unfortunately, some time later, the boss comes in and asks us to implement filtering by size, too. So we jump back into `ProductFilter.cs`, add the following code, and recompile:

```
public IEnumerable<Product> FilterBySize
  (IEnumerable<Product> products, Size size)
{
  foreach (var p in products)
    if (p.Size == size)
      yield return p;
}
```

This feels like outright duplication, doesn't it? Why don't we just write a general method that takes a predicate (i.e., a `Predicate<T>`)? Well, one reason could be that different forms of filtering can be done in different ways: for example, some record types might be indexed and need to be searched in a specific way; some data types are amenable to search on a Graphics Processing Unit (GPU), while others are not.

Furthermore, you might want to restrict the criteria one can filter on. For example, if you look at Amazon or a similar online store, you are only allowed to perform filtering on a finite set of criteria. Those criteria can be added or removed by Amazon if they find that, say, sorting by number of reviews interferes with the bottom line.

Okay, so our code goes into production but, once again, the boss comes back and tells us that now there's a need to search by both size *and* color. So what are we to do but add another function?

```
public IEnumerable<Product> FilterBySizeAndColor(
  IEnumerable<Product> products,
  Size size, Color color)
{
  foreach (var p in products)
    if (p.Size == size && p.Color == color)
      yield return p;
}
```

What we want, from the preceding scenario, is to enforce the *Open-Closed Principle* that states that a type is open for extension but closed for modification. In other words, we want filtering that is extensible (perhaps in a different assembly) without having to modify it (and recompiling something that already works and may have been shipped to clients).

How can we achieve it? Well, first of all, we conceptually separate (SRP!) our filtering process into two parts: a filter (a construct which takes all items and only returns some) and a specification (a predicate to apply to a data element).

We can make a very simple definition of a specification interface:

```
public interface ISpecification<T>
{
  bool IsSatisfied(T item);
}
```

In this interface, type `T` is whatever we choose it to be: it can certainly be a `Product`, but it can also be something else. This makes the entire approach reusable.

Next up, we need a way of filtering based on an `ISpecification<T>`; this is done by defining, you guessed it, an `IFilter<T>`:

```
public interface IFilter<T>
{
  IEnumerable<T> Filter(IEnumerable<T> items,
                        ISpecification<T> spec);
}
```

Again, all we are doing is specifying the signature for a method called `Filter()` which takes all the items and a specification and returns only those items that conform to the specification.

Based on the preceding data, the implementation of an improved filter is really simple:

```
public class BetterFilter : IFilter<Product>
{
  public IEnumerable<Product> Filter(IEnumerable<Product> items,
                                     ISpecification<Product> spec)
  {
    foreach (var i in items)
```

```
      if (spec.IsSatisfied(i))
        yield return i;
  }
}
```

Again, you can think of an `ISpecification<T>` that's being passed in as a strongly typed equivalent of a `Predicate<T>` that has a finite set of concrete implementations suitable for the problem domain.

Now, here's the easy part. To make a color filter, you make a `ColorSpecification`:

```
public class ColorSpecification : ISpecification<Product>
{
  private Color color;

  public ColorSpecification(Color color)
  {
    this.color = color;
  }

  public bool IsSatisfied(Product p)
  {
    return p.Color == color;
  }
}
```

Armed with this specification, and given a list of products, we can now filter them as follows:

```
var apple = new Product("Apple", Color.Green, Size.Small);
var tree = new Product("Tree", Color.Green, Size.Large);
var house = new Product("House", Color.Blue, Size.Large);

Product[] products = {apple, tree, house};

var pf = new ProductFilter();
WriteLine("Green products:");
foreach (var p in pf.FilterByColor(products, Color.Green))
  WriteLine($" - {p.Name} is green");
```

The preceding code gets us "Apple" and "Tree" because they are both green. Now, the only thing we haven't implemented so far is searching for size *and* color (or, indeed, explained how you would search for size *or* color, or mix different criteria). The answer is that you simply make a *combinator*. For example, for the logical AND, you can make it as follows:

```
public class AndSpecification<T> : ISpecification<T>
{
  private readonly ISpecification<T> first, second;

  public AndSpecification(ISpecification<T> first, ISpecification<T> second)
  {
    this.first = first;
    this.second = second;
  }

  public override bool IsSatisfied(T t)
  {
    return first.IsSatisfied(t) && second.IsSatisfied(t);
  }
}
```

And now, you are free to create composite conditions on the basis of simpler `ISpecifications`. Reusing the `green` specification we made earlier, finding something green and big is now as simple as

```
foreach (var p in bf.Filter(products,
  new AndSpecification<Product>(
    new ColorSpecification(Color.Green),
    new SizeSpecification(Size.Large))))
{
  WriteLine($"{p.Name} is large and green");
}

// Tree is large and green
```

This was a lot of code to do something seemingly simple, but the benefits are well worth it. The only really annoying part is having to specify the generic argument to `AndSpecification` – remember, unlike the color/size specifications, the combinator isn't constrained to the `Product` type.

Keep in mind that, thanks to the power of C#, you can simply introduce an `operator &` (important: single ampersand here, && is a byproduct) for two `ISpecification<T>` objects, thereby making the process of filtering by two (or more!) criteria somewhat simpler… the only problem is that we need to change from an interface to an abstract class (feel free to remove the leading `I` from the name).

```
public abstract class ISpecification<T>
{
  public abstract bool IsSatisfied(T p);

  public static ISpecification<T> operator &(
    ISpecification<T> first, ISpecification<T> second)
  {
    return new AndSpecification<T>(first, second);
  }
}
```

If you now avoid making extra variables for size/color specifications, the composite specification can be reduced to a single line[3]:

```
var largeGreenSpec = new ColorSpecification(Color.Green)
                   & new SizeSpecification(Size.Large);
```

Naturally, you can take this approach to extreme by defining extension methods on all pairs of possible specifications:

```
public static class CriteriaExtensions
{
  public static AndSpecification<Product> And(this Color color, Size size)
  {
    return new AndSpecification<Product>(
      new ColorSpecification(color),
      new SizeSpecification(size));
  }
}
```

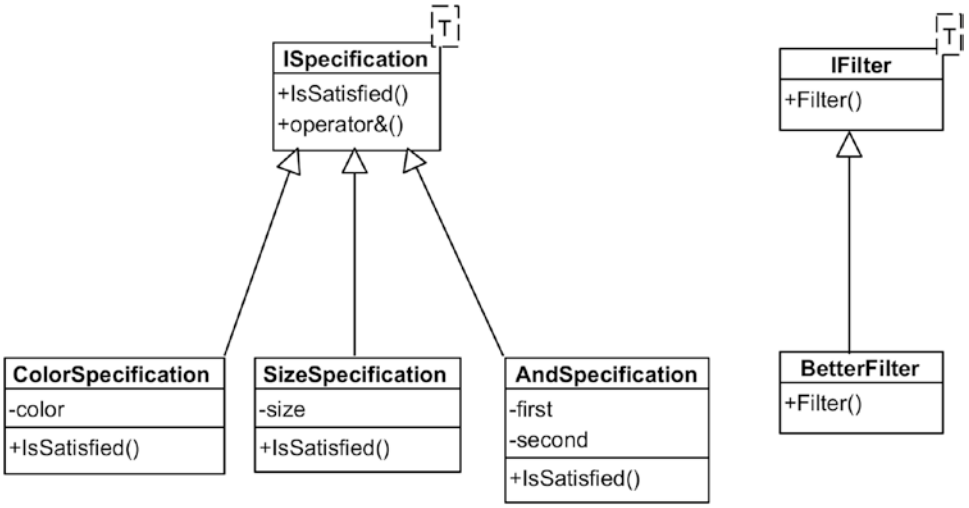
[3]Notice we're using a single & in the evaluation. If you want to use &&, you'll also need to override the `true` and `false` operators in `ISpecification`.

with the subsequent use

```
var largeGreenSpec = Color.Green.And(Size.Large);
```

However, this would require a set of pairs of all possible criteria, something that's not particularly realistic, unless you use code generation, of course. Sadly, there is no way in C# of establishing an implicit relationship between an `enum Xxx` and an `XxxSpecification`.

Here is a diagram of the entire system we've just built:



So, let's recap what OCP is and how this example enforces it. Basically, OCP states that you shouldn't need to go back to code you've already written and tested and change it. And that's exactly what's happening here! We made `ISpecification<T>` and `IFilter<T>` and, from then on, all we have to do is implement either of the interfaces (without modifying the interfaces themselves) to implement new filtering mechanics. This is what is meant by "open for extension, closed for modification."

One thing worth noting is that conformance with OCP is only possible inside an object-oriented paradigm. For example, F#'s discriminated unions are by definition not compliant with OCP since it is impossible to extend them without modifying their original definition.

Liskov Substitution Principle

The Liskov Substitution Principle, named after Barbara Liskov, states that if an interface takes an object of type `Parent`, it should equally take an object of `type Child` without anything breaking. Let's take a look at a situation where LSP is broken.

Here's a rectangle; it has width and height and a bunch of getters and setters calculating the area:

```
public class Rectangle
{
  public int Width { get; set; }
  public int Height { get; set; }

  public Rectangle() {}
  public Rectangle(int width, int height)
  {
    Width = width;
    Height = height;
  }

  public int Area => Width * Height;
}
```

Suppose we make a special kind of `Rectangle` called a `Square`. This object overrides the setters to set both width *and* height:

```
public class Square : Rectangle
{
  public Square(int side)
  {
    Width = Height = side;
  }

  public new int Width
  {
    set { base.Width = base.Height = value; }
  }
```

```
  public new int Height
  {
    set { base.Width = base.Height = value; }
  }
}
```

This approach is *evil*. You cannot see it yet, because it looks very innocent indeed: the setters simply set both dimensions (so that a square always remain a square), what could possibly go wrong? Well, suppose we introduce a method that makes use of a `Rectangle`:

```
public static void UseIt(Rectangle r)
{
  r.Height = 10;
  WriteLine($"Expected area of {10*r.Width}, got {r.Area}");
}
```

This method looks innocent enough if used with a `Rectangle`:

```
var rc = new Rectangle(2,3);
UseIt(rc);
// Expected area of 20, got 20
```

However, innocuous method can seriously backfire if used with a `Square` instead:

```
var sq = new Square(5);
UseIt(sq);
// Expected area of 50, got 100
```

The preceding code takes the formula `Area = Width × Height` as an invariant. It gets the width, sets the height to 10, and rightly expects the product to be equal to the calculated area. But calling the preceding function with a `Square` yields a value of 100 instead of 50. I'm sure you can guess why this is.

So the problem here is that although `UseIt()` is happy to take any `Rectangle` class, it fails to take a `Square` because the behaviors inside `Square` break its operation. So, how would you fix this issue? Well, one approach would be to simply deprecate the `Square` class and start treating some `Rectangles` as special case. For example, you could introduce an `IsSquare` property.

You might also want a way of detecting that a `Rectangle` is, in fact, a square:

```
public bool IsSquare => Width == Height;
```

Similarly, instead of having constructors, you could introduce Factory Methods (see the "Factories" chapter) that would construct rectangles and squares and would have corresponding names (e.g., `NewRectangle()` and `NewSquare()`), so there would be no ambiguity.

As far as setting the properties is concerned, in this case, the solution would be to introduce a uniform `SetSize(width,height)` method and remove `Width/Height` setters entirely. That way, you are avoiding the situation where setting the height via a setter also stealthily changes the width.

This rectangle/square challenge is, in my opinion, an excellent interview question: it doesn't have a correct answer, but allows many interpretations and variations.

Interface Segregation Principle

Oh-kay, here is another contrived example that is nonetheless suitable for illustrating the problem. Suppose you decide to define a multi-function printer: a device that can print, scan, and also fax documents. So you define it like so:

```
class MyFavouritePrinter /* : IMachine */
{
  void Print(Document d) {}
  void Fax(Document d) {}
  void Scan(Document d) {}
};
```

This is fine. Now, suppose you decide to define an interface that needs to be implemented by everyone who also plans to make a multi-function printer. So you could use the Extract Interface function in your favorite IDE, and you'll get something like the following:

```
public interface IMachine
{
  void Print(Document d);
  void Fax(Document d);
  void Scan(Document d);
}
```