

Design Patterns in .NET

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

Dmitri Nesteruk

Apress®

Design Patterns in .NET

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

Dmitri Nesteruk

Apress®

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

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

ISBN-13 (pbk): 978-1-4842-4365-7
<https://doi.org/10.1007/978-1-4842-4366-4>

ISBN-13 (electronic): 978-1-4842-4366-4

Copyright © 2019 by Dmitri Nesteruk

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

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

Cover designed by eStudioCalamar

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 rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

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/9781484243657. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

Table of Contents

About the Authorxi

Introductionxiii

Part I: Introduction 1

Chapter 1: The SOLID Design Principles3

 Single Responsibility Principle.....3

 Open-Closed Principle.....6

 Liskov Substitution Principle.....14

 Interface Segregation Principle17

 Dependency Inversion Principle.....22

Chapter 2: The Functional Perspective27

 Function Basics.....27

 Functional Literals in C#30

 Storing Functions in C#.....30

 Functional Literals in F#.....33

 Composition35

 Functional-Related Language Features36

Part II: Creational Patterns 39

Chapter 3: Builder.....41

 Scenario.....41

 Simple Builder.....44

TABLE OF CONTENTS

Fluent Builder.....	45
Communicating Intent.....	46
Composite Builder.....	48
Builder Parameter.....	52
Fluent Interface Inheritance.....	55
DSL Construction in F#.....	59
Summary.....	61
Chapter 4: Factories	63
Scenario.....	63
Factory Method.....	65
Factory	67
Inner Factory.....	67
Logical Separation.....	69
Abstract Factory.....	69
Functional Factory	72
Summary.....	74
Chapter 5: Prototype.....	77
Deep vs. Shallow Copying.....	77
ICloneable Is Bad	78
Deep Copying with a Special Interface	80
Deep Copying Objects.....	81
Duplication via Copy Construction	83
Serialization	84
Prototype Factory.....	86
Summary.....	88

Chapter 6: Singleton	91
Singleton by Convention	92
Classic Implementation.....	93
Lazy Loading.....	94
The Trouble with Singleton.....	95
Singletons and Inversion of Control	100
Monostate	101
Summary.....	102
Part III: Structural Patterns	103
Chapter 7: Adapter	105
Scenario.....	105
Adapter	107
Adapter Temporaries.....	109
The Problem with Hashing.....	113
Property Adapter (Surrogate)	116
Adapters in the .NET Framework	118
Summary.....	119
Chapter 8: Bridge.....	121
Conventional Bridge.....	121
Dynamic Prototyping Bridge	125
Summary.....	129
Chapter 9: Composite	131
Grouping Graphic Objects	131
Neural Networks	134
Shrink Wrapping the Composite.....	138
Summary.....	140

TABLE OF CONTENTS

Chapter 10: Decorator	141
Custom String Builder	141
Adapter-Decorator.....	144
Multiple Inheritance	145
Dynamic Decorator Composition.....	149
Static Decorator	153
Functional Decorator.....	155
Summary.....	156
Chapter 11: Façade.....	159
Building a Trading Terminal.....	161
An Advanced Terminal.....	162
Where's the Façade?	165
Summary.....	167
Chapter 12: Flyweight	169
Usernames	169
Text Formatting	172
Summary.....	176
Chapter 13: Proxy	177
Protection Proxy	177
Property Proxy.....	180
Virtual Proxy	183
Communication Proxy	186
Summary.....	189

Part IV: Behavioral Patterns.....	191
Chapter 14: Chain of Responsibility	193
Scenario.....	193
Method Chain.....	194
Broker Chain	198
Summary.....	203
Chapter 15: Command	205
Scenario.....	205
Implementing the Command Pattern	206
Undo Operations.....	208
Composite Commands	212
Functional Command	216
Queries and Command Query Separation	218
Summary.....	219
Chapter 16: Interpreter	221
Numeric Expression Evaluator	222
Lexing	223
Parsing	226
Using Lexer and Parser	230
Interpretation in the Functional Paradigm	230
Summary.....	235
Chapter 17: Iterator	237
Array-Backed Properties.....	238
Let's Make an Iterator	241
Improved Iteration.....	245
Summary.....	247

TABLE OF CONTENTS

Chapter 18: Mediator.....249

 Chat Room..... 249

 Mediator with Events 254

 Summary..... 259

Chapter 19: Memento261

 Bank Account..... 261

 Undo and Redo..... 263

 Summary..... 267

Chapter 20: Null Object.....269

 Scenario 269

 Intrusive Approach 271

 Null Object..... 271

 Design Improvements 272

 Null Object Virtual Proxy..... 273

 Dynamic Null Object..... 274

 Summary..... 276

Chapter 21: Observer.....279

 Weak Event Pattern..... 281

 Property Observers 284

 Dependency Problems 286

 Event Streams..... 292

 Observable Collections..... 297

 Declarative Subscriptions 298

 Summary..... 300

Chapter 22: State	301
State-Driven State Transitions	302
Handmade State Machine	305
State Machines with Stateless	309
Types, Actions, and Ignoring Transitions	309
Reentrancy Again	311
Hierarchical States	312
More Features	312
Summary	314
Chapter 23: Strategy	315
Dynamic Strategy	316
Static Strategy	319
Functional Strategy	321
Summary	322
Chapter 24: Template Method	323
Game Simulation	323
Functional Template Method	326
Summary	328
Chapter 25: Visitor	329
Intrusive Visitor	330
Reflective Printer	332
Functional Reflective Visitor	334
Improvements	335
What Is Dispatch?	336
Dynamic Visitor	338

TABLE OF CONTENTS

Classic Visitor340

 Implementing an Additional Visitor343

Acyclic Visitor345

Functional Visitor348

Summary.....349

Index.....351

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 field-programmable gate arrays (FPGAs). He has been a C# MVP since 2009.

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 that aren't even properly object-oriented programming (OOP)! So why another book on it? I know that if you're reading this in a bookstore, you probably have a limited amount of time to decide whether this is worth the investment.

I decided to write this book to fill a gap left by the lack of patterns books in the .NET space. Plenty of books have been written over the years, but not one has attempted to research all the ways in which modern C# and F# language features can be used to implement design patterns and 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 book² 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

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

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

INTRODUCTION

or less essential and unavoidable, whereas other patterns are more of a scientific curiosity (but nevertheless are discussed in this book, because 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 necessary for most typical scenarios. Although overengineering is a lot of fun (hey, you get to really solve the problem and impress your coworkers), it's often not feasible due to time, cost, and complexity constraints.

Who This Book Is For

This book is designed to be a modern-day update to the classic Gang of Four (GoF, referring to the four authors) 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 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., `dynamic`) 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 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.

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 reference other programming languages such as C++ or Kotlin. It is 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 mention those when we come to them.

PART I

Introduction

CHAPTER 1

The SOLID Design Principles

SOLID is an acronym that 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 Martin'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 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;
}

```

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 here). This is easy:

```

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

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

```

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 to you having to make lots of tiny changes in lots of classes, whether related (as in a hierarchy) or not, is typically a *code smell*—an indication that something’s not quite right. Now, it really depends on the situation: If you’re renaming a symbol that is being used in a hundred places, I would argue that’s generally okay because ReSharper, Rider, or whatever integrated development environment (IDE) you use will actually let you perform a refactoring and have the change propagate everywhere. When you need to completely rework an interface, though, it can be 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) 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());
    }
}
```

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 is 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-pattern¹ that 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 methods) and, thanks to continuous integration and source control systems, the responsible developer can be quickly identified and adequately punished.

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
}
```

¹An *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 is difficult to understand, maintain, and refactor.

```

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, although of course it could be greatly simplified with the use of LINQ. So, our code goes into production but, unfortunately, some time later, the

boss 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 units (GPU) whereas 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 is a need to search by both size and color. So what are we to do but add another methods?

```
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 this 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 might 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 that 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 code, 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 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() that takes all the items and a specification, and returns only those items that conform to the specification.

Based on these interface definitions, 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 is 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");
```

This code gets us “Apple” and “Tree” because they are both green. Now, the only thing we have not implemented so far is searching for size *and* color (or, indeed, explaining how you would search for size *or* color, or mix different criteria). The answer is that you simply make a *composite* specification (or 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);
    }
}
```

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 this:

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

// 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 and size specifications, the combinator is not constrained to the Product type.

Keep in mind that, thanks to the power of C#, you can simply introduce an operator & (important: note the single ampersand here; && is a by-product) 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 and color specifications, the composite specification can be reduced to a single line:²

```
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));
    }
}
```

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 is 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`.

Figure 1-1 is a diagram of the entire system we've just built.

²Notice 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`.

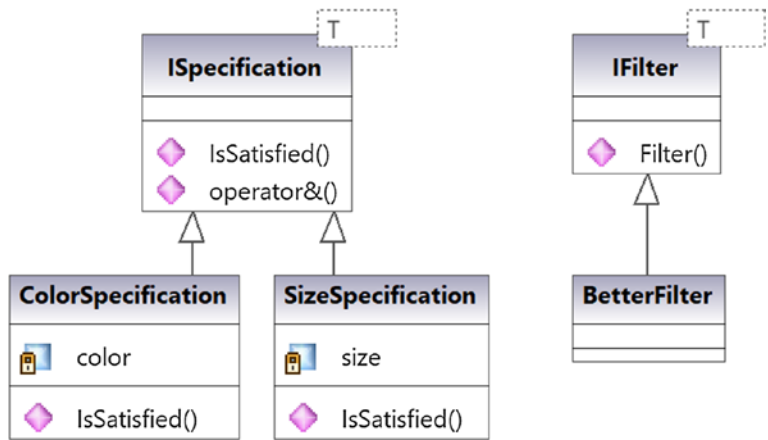


Figure 1-1. Visual representation of the system built

So, let’s recap what OCP is and how the given example enforces it. Basically, OCP states that you shouldn’t need to go back to code you have already written and tested and change it. That is 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 because 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, and a property getter for 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 remains 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, this 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 $\text{Area} = \text{Width} \times \text{Height}$ as an invariant. It gets the width, sets the height to 10, and rightly expects the product to be equal to the calculated area. Calling this method with a `Square` yields a value of 100 instead of 50. I'm sure you can guess why this is.