Design Patterns in C#

Dmitri Nesteruk

Design Patterns in C#

Dmitri Nesteruk

This book is for sale at http://leanpub.com/csharp_patterns

This version was published on 2020-05-13

Contents

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 programming (OOP)! 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 GoF 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 focussed 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 2nd 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 ‘non-working month’, a curious occurence 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 round 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.

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 5 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;
}
```

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 below). 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 lost 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[3]) when talking about the general approach of splitting code into separate classes by functionality. In the cases of perisistence 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[4] 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

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

[4]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 is an anti-pattern is that anti-patterns are typically patterns of bad design, resulting in code that's difficult to understand, maintain and refactor.

(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.

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:

```
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 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 code below 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 above scenario, is to enfoce 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 above, 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 `ColorSpecifica-tion`:

```
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 above 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 earier, 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");
}

// 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:[5]

```
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));
  }
}
```

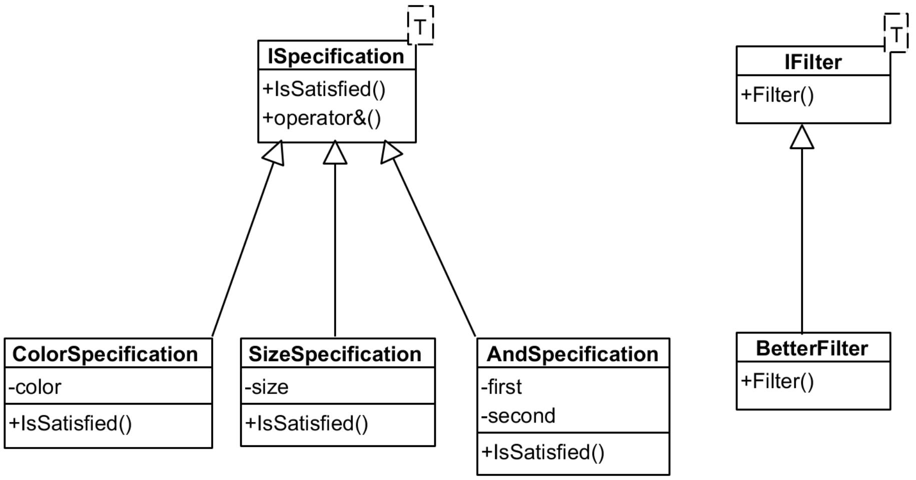
with the subsequent use:

[5]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`.

```
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 code above takes the formula `Area = Width Ã— 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 above 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 avoid 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 multifunction 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 multifunction printer. So you could use the Extract Interface function in your favourite IDE and you'll get something like the following:

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

This is a problem. The reason it is a problem is that some implementor of this interface might not need scanning or faxing, just printing. And yet, you are forcing them to implement those extra features: sure, they can all be no-op, but why bother with this?

A typical example would be a good old-fashioned printer that doesn't have any scanning or fax features. Implementing the `IMachine` interface in this situation becomes a real challenge. What's particularly frustrating about this situation is there is no *correct* way of leaving things unimplemented. I mean, sure, you can throw an exception, and we even have a dedicated exception precisely for this purpose:

```
public class OldFashionedPrinter : IMachine
{
  public void Print(Document d)
  {
    // yep
  }

  public void Fax(Document d)
  {
    throw new System.NotImplementedException();
  }

  public void Scan(Document d)
  {
    throw new System.NotImplementedException();
  }
}
```

But you are still confusing the user! They can see `OldFashionedPrinter.Fax()` as part of the API, so they can be forgiven for thinking that this type of printer can fax, too! So what else can you do? Well, you can just leave the extra methods as no-op (empty), just like the `Scan()` method above. Again, this approach violates the *Principle of Least Surprise*: your users want things to be as predictable as you can possibly make them. And neither a method that throws by default, nor a method that does nothing is the most predictable solution – even if you make it explicit in the documentation!

The only option that would categorically work at compile-time is the nuclear option of marking all unnecessary methods obsolete:

```
[Obsolete("Not supported", true)]
public void Scan(Document d)
{
  throw new System.NotImplementedException();
}
```

This will prevent compilation if someone does try to use `OldFashionedPrinter.Scan()`. In fact, good IDEs will recognize this ahead of time, and will often cross out the method as you call it to indicate that it's not going to work. The only issue with this approach is that it's deeply unidiomatic: the method isn't really obsolete, it's unimplemented. Stop lying to the client!

So what the Interface Segregation Principle suggests you do instead is split up interfaces, so that implementors can pick and choose depending on their needs. Since printing and scanning are different operations (for example, a Scanner cannot print), we define separate interfaces for these:

```
public interface IPrinter
{
  void Print(Document d);
}

public interface IScanner
{
  void Scan(Document d);
}
```

Then, a printer can implement *just* the required functionality, nothing else:

```
public class Printer : IPrinter
{
  public void Print(Document d)
  {
    // implementation here
  }
}
```

Similarly, if we want to implement a photocopier, we can do so by implementing the `IPrinter` and `IScanner` interfaces:

```
public class Photocopier : IPrinter, IScanner
{
  public void Print(Document d) { ... }
  public void Scan(Document d) { ... }
}
```

Now, if we really want a dedicated interface for a multi-function device, we can define it as a combination of the aforementioned interfaces:

```
public interface IMultiFunctionDevice
  : IPrinter, IScanner // also IFax etc.
{
  // nothing here
}
```

And when you make a class for a multifunction device, this is the interface to use. For example, you could use simple delegation to ensure that `Machine` reuses the functionality provided by a particular `IPrinter` and `IScanner` (this is actually a good illustration of the Decorator pattern):

```
public class MultiFunctionMachine : IMultiFunctionDevice
{
  // compose this out of several modules
  private IPrinter printer;
  private IScanner scanner;

  public MultiFunctionMachine(IPrinter printer, IScanner scanner)
  {
    this.printer = printer;
    this.scanner = scanner;
  }

  public void Print(Document d)
  {
    printer.Print(d);
  }

  public void Scan(Document d)
  {
    scanner.Scan(d);
  }
}
```

So, just to recap, the idea here is to segregate parts of a complicated interface into separate interfaces so as to avoid forcing clients to implement functionality that they do not really need. Anytime when you write a plugin for some complicated application and you're given an interface with 20 confusing methods to implement with various no-ops and `return null`s, more likely than not the API authors have violated the ISP.

Parameter Object

When we talk about interfaces, we typically talk about the `interface` keyword, but the essence of ISP can also be applied to a much more local phenomenon: interfaces in the conventional sense, e.g., the parameter list exposed by a constructor.

Consider a (completely arbitrary) example of a constructor that takes a large number of parameters. Most of these parameters have defaults, but some do not:

```
public class Foo
{
  public Foo(int a, int b, bool c = false, int d = 42, float e = 1.0f)
  {
    // meaningful code here
  }
}
```

The problem with the interface of the constructor above is that it throws a lot into the face of an unsuspecting client. The situation becomes even more comical if the client has to provide arguments `a`, `b` and `e` because then they'll end up repeating some of the defaults unnecessarily.

In this situation, the core principle of ISP (do not throw everything into an interface) also makes sense here, though for different reasons. You need to provide a sensible set of inputs and let the user avoid the fuss associated with any extras.

Any self-respecting IDE offers you the **Parameter Object** refactoring – an ability to take all parameters and put them into a class with all the defaults preserved:

```
public class MyParams
{
  public int a;
  public int b;
  public bool c = false;
  public int d = 42;
  public float e = 1.0f;

  public MyParams(int a, int b)
  {
    this.a = a;
    this.b = b;
  }
}
```

This parameter object would then be passed into `Foo`'s constructor:

```
public Foo(MyParams myParams)
{
  // meaningful work here
}
```

Notice how `MyParams` is crafted: it does have a constructor of its own, mandating that you initialize the first two parameters, but it also exposes other parameters for you to initialize arbitrarily.

All I'm trying to say is this: principles and patterns don't have to operate at the macro (class) scale – they are also good enough to operate on the micro scales.

Dependency Inversion Principle

The original definition of the Dependency Inversion Principle states the following[6]:

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

What this statement basically means is that, if you're interested in logging, your reporting component should not depend on a concrete `ConsoleLogger`, but can depend on an `ILogger` interface. In this case, we are considering the reporting component to be high-level (closer to the business domain), whereas logging, being a fundamental concern (kind of like file I/O or threading, but not quite) is considered a low-level module.

B. Abstractions should not depend on details. Details should depend on abstractions.

This is, once again, restating that dependencies on interfaces or base classes is better than dependencies on concrete types. Hopefully the truth of this statement is obvious, because such an approach supports better configurability and testability… especially you're using a good framework to handle these dependencies for you.

Let's take a look at an example of DIP in action. Suppose we decide to model genealogical relationship between people using the following definitions:

[6]Martin, Robert C. (2003), *Agile Software Development, Principles, Patterns, and Practices*, Prentice Hall, pp. 127â€“131.

```
public enum Relationship
{
  Parent,
  Child,
  Sibling
}

public class Person
{
  public string Name;
  // DoB and other useful properties here
}
```

We create a (low-level) class specifically for storing information about relationships. It would look something like the following:

```
public class Relationships // low-level
{
  public List<(Person,Relationship,Person)> relations
    = new List<(Person, Relationship, Person)>();

  public void AddParentAndChild(Person parent, Person child)
  {
    relations.Add((parent, Relationship.Parent, child));
    relations.Add((child, Relationship.Child, parent));
  }
}
```

Now, suppose we want to do some research on the relationships we've captured. For example, in order to find all the children of John, we create the following (high-level) class:

```
public class Research
{
  public Research(Relationships relationships)
  {
    // high-level: find all of john's children
    var relations = relationships.Relations;
    foreach (var r in relations
      .Where(x => x.Item1.Name == "John"
                  && x.Item2 == Relationship.Parent))
    {
      WriteLine($"John has a child called {r.Item3.Name}");
    }
  }
}
```

The approach illustrated above directly violates DIP because a high-level module `Research` directly depends on the low-level module `Relationships`. Why is this bad? Because `Research` depends directly on the data storage implementation of `Relationships`: you can see it iterating the list of tuples. What if you wanted to later change the underlying storage of `Relationships`, perhaps by moving it from a list-of-tuples to a proper database? Well, you couldn't, because you have high-level modules depending on it.

So what do we want? We want our high-level module to depend on an *abstraction* which, in C# terms means depending on an interface of some kind. But we don't have an interface yet! No problem, let's create one:

```
public interface IRelationshipBrowser
{
  IEnumerable<Person> FindAllChildrenOf(string name);
}
```

This interface has a single method for finding all children of a particular person by name. We expect that a low-level module such as `Relashionships` would be able to implement this method and thereby keep its implementation details private:

```
public class Relationships : IRelationshipBrowser // low-level
{
  // no longer public!
  private List<(Person,Relationship,Person)> relations
    = new List<(Person, Relationship, Person)>();

  public IEnumerable<Person> FindAllChildrenOf(string name)
  {
    return relations
      .Where(x => x.Item1.Name == name
               && x.Item2 == Relationship.Parent)
      .Select(r => r.Item3);
  }
}
```

Now this is something that our `Research` module can depend upon! We can inject an `IRelationshipBrowser` into its constructor and perform the research safely, without digging into the low level module's internals:

```
public Research(IRelationshipBrowser browser)
{
  foreach (var p in browser.FindAllChildrenOf("John"))
  {
    WriteLine($"John has a child called {p.Name}");
  }
}
```

Please note that DIP isn't the equivalent of dependency *injection*, which is another important topic in its own right. DI can facilitate the application of DIP by simplifying the representation of dependencies, but those two are separate concepts.

The Functional Perspective

The functional paradigm is supported by both the C# and F# languages. Both languages can claim to be multi-paradigm since they fully support both OOP and functional programming, though F# has more of a ‘functional first’ mindset with object-orientation added for completeness, whereas in C# the integration of functional programming aspects appears to be much more harmonious.

Here we are going to take a very cursory look at functional programming in the C# and F# languages. Some of the material may already be familiar to you; in that case, feel free to skip this part.

Function Basics

First, a note on notation. In this book, I use the words *method* and *function* interchangeably to mean the same thing: a self-contained operation that takes zero or more inputs and has zero or more outputs (return values). I will use the word *method* when working in the C# domain, and likewise will use the word *function* when dealing with the functional domain.

In C#, functions are not freestanding: they must be members of some class or other. For example, to define integer addition, you must pack the `Add()` method into some class (let’s call it `Ops`):

```
class Ops
{
  public static int Add(int a, int b)
  {
    return a + b;
  }
}
```

This function is meant to be called as `Ops.Add()` though you can shorten it to just `Add()` if you use C#’s `import static` instruction. Still, this is a particular pain point for the mathematicians because, even if you add `using static System.Math;` to ever single file in your project, you still end up having to use uppercase names for functions like `Sin()` — not an ideal situation!

In F#, the approach is drastically different. The above addition function can be defined as:

```
let add a b = a + b
```

It may appear as if some magic has happened: we didn't define a class, nor did we specify the data types of arguments. And yet, if you were to look at the C#-equivalent code, you would see something like the following:

```
[CompilationMapping]
public static class Program
{
  [CompilationArgumentCounts(new int[] {1, 1})]
  public static int add(int a, int b)
  {
    return a + b;
  }
}
```

As you may have guessed, the static class `Program` got its name from the name of the file the code was in (in this case, `Program.fs`). The types of arguments were chosen as a guesstimate. What if we were to add a call with different argument types?

```
let ac = add "abra" "cadabra"
printfn "%s" ac
```

The above code prints "abracadabra", of course, but what's interesting is the code generated… you've guessed it already, haven't you?

```
[CompilationArgumentCounts(new int[] {1, 1})]
public static string add(string a, string b)
{
  return a + b;
}
```

The reason why this is possible is called *type inference*: the compiler figures out which types you're actually using in a function, and tries to accommodate by constructing a function with corresponding parameters. Sadly, this is not a silver bullet. For example, if you were to subsequently add another call – this time, with integers – it would fail:

```
let n = add 1 2
// Error: This expression was expected to have type 'string' but here has type \
'int'
```

Functional Literals in C#

It's not always convenient to define functions inside classes: sometimes you want to create a function exactly where you need it, i.e., in another function. These sorts of functions are called *anonymous* because they are not given persistent names; instead, the function is stored in a delegate.

The old-fashioned, C# 2.0 way of defining anonymous functions is with the use of a `delegate` keyword, similar to the following:

```
BinaryOperation multiply = delegate(int a, int b) { return a * b; };
int x = multiply(2, 3); // 6
```

Of course, since C# 3.0 we have a much more convenient way of defining the same thing.

```
BinaryOperation multiply = (a, b) => { return a * b; };
```

Notice the disappearance of type information next to a and b: this is type inference at work once again!

Finally, since C# 6 we have expression-bodied members which allow us to get omit the `return` keyword in single-statement evaluations, shortening the definition to the following:

```
BinaryOperation multiply = (a, b) => a + b;
```

Of course, anonymous functions are useless if you don't store them somewhere, and as soon as you're storing something, that something needs a *type*. Luckily, we have types of this, too.

Storing Functions in C#

A key feature of functional programming is being able to refer to functions and call them through references. In C#, the simplest way to do this is using delegates.

A *delegate type* is to a function what a class is to an instance. Given our `Add()` function from above, we can define a delegate similar to the following:

```
public delegate int BinaryOperation(int a, int b);
```

A delegate doesn't have to live inside a C# class: it can exist at a namespace level. So, in a way, you can treat it as a type declaration. Of course, you can also stick a delegate into a class, in which case you can treat it as a *nested* type declaration.

Having a delegate such as this lets us store a reference to a function in a variable:

```
BinaryOperation op = Ops.Add;
int x = op(2, 3);
```

Compared to instances of a class, there's a note that needs to be made here — not only does a delegate instance know *which* function needs to be called, but it also knows the *instance* of the class on which this method should be called. This distinction is critical because it allows us to distinguish, for example, static and non-static functions.

Any other function, which has the same signature can also be assigned to this delegate, regardless of who is its logical owner. For example, you could define a function called `Subtract()` virtually anywhere and assign it to the delegate. This includes defining it as an ordinary member function…

```
class Program
{
  static int Subtract(int a, int b) => a - b;
  static void Main(string[] args)
  {
      BinaryOperation op = Subtract;
      int x = op(10, 2); // 8
  }
}
```

However, it can easily be a local (nested) function:

```
static void Main(string[] args)
{
  int Multiply(int a, int b) => a * b;
  BinaryOperation op = Multiply;
  int x = op(10, 2); // 20
}
```

…or even an anonymous delegate or a lambda function:

```
void SomeMethod()
{
  BinaryOperation op = (a, b) => a / b;
  int x = op(10, 2); // 5
}
```

Now, here's the important part, pay attention: in the majority of cases, *defining your own delegates is not necessary*. Why? Because the .NET Base Class Library (BCL) comes with predefined delegates of up to 16 parameters in length (C# has no variadic templates[7]) which cover most cases that you might be interested in.

The `Action` delegate represents a function that doesn't return a value (is `void`). Its generic arguments relate to the types of arguments this function takes. So you can write something like:

[7]Variadic templates are primarily a C++ concept. They allow you to define template (generic) types and methods that take an *arbitrary* number of type arguments, and provide syntax for efficiently iterating the argument type list. .NET generics are implemented differently to C++ templates (their 'genericity' is preserved at runtime), so variadics in .NET are not possible.

```
Action doStuff = () => Console.WriteLine("doing stuff!");
doStuff(); // prints "doing stuff!"

Action<string> printText = x => Console.WriteLine(x);
printText("hello"); // prints "hello"
```

The generic arguments of `Action` are needed to specify parameter types. If a function takes no parameters, just use a non-generic `Action`.

If your function does need to return a value, then you can use a predefined delegate `Func<T1, T2, ..., TR>`. This is always generic, where TR has the type of the return value. In our case, we could have defined a binary operation as:

```
Func<int, int, int> mul = Multiply;
// or
Func<int, int, int> div = (a, b) => a / b;
```

Together, `Action` and `Func` cover all the realistic needs you might encounter for a delegate. Sadly, these delegates themselves cannot be deduced through type inference. In other words, you cannot write

```
var div = (int a, int b) => a / b;
```

expecting `div` to be of type `Func<int, int, int>` — this simply will not compile.

Functional Literals in F#

In F#, the process of defining a function is a lot more harmonized. For example, there is no real distinction between the syntax for defining a variable and that of defining a method on the global scope.

```
let add a b = a + b

[<EntryPoint>]
let main argv =
  let z = add
  let result = z 1 2
  0
```

However, the decompiled results of this code are too frightening to show here. What is important to realize is that F# *does*, in fact, automatically map your function to a type without any extra hints. But instead of mapping it to a `Func` delegate, it maps it to its own type called `FSharpFunc`.

In order to understand the reason for `FSharpFunc`'s existence, we need to understand something called *currying*. Currying (nothing to do with Indian food) is an entirely different approach to the way functions are defined and called. Remember when our F# function `add a b` got turned into a C# equivalent `int add(int a, int b)`? Well let me show you a very similar situation where this will *not* happen:

```
let printValues a b =
  printf "a = %i; b = %i" a b
```

What does this compile to? Well, without showing extra levels of gore, the compiler generates, among other things, a class inheriting from `FSharpFunc<int, Unit>` (`Unit` can be seen as F#'s equivalent of `void`) that also happens to have another `FSharpFunc<int, Unit>` as an invocable member. Why?!?

Well, to simplify things, your `printValues` call actually got turned into something like

```
let printValues a =
  let printValues@10-1 b =
    printf "a = %i; b = %i" a b
  return printValues@10-1
```

So, in simplified C# terms, instead of making a function callable as `printValues(a,b)`, we made a function callable as `printValues(a)(b)`.

What's the advantage of this? Well, let's come back to our `add` function:

```
let add a b = a + b
```

We can now use this function to define a new function called `addFive` that adds 5 to a given number. This function can be defined as follows:

```
let addFive x = add 5 x
```

We can now call it as:

```
let z = addFive 5 // z = 10
```

Having this definition forces the compile to express the invocation of any call of `add x y` as being equivalent to `add(x)(y)`. But `add(x)` (without the `y`) is already prepackaged as a standalone `FSharpFunc<int,int>` that itself yields a function that takes a `y` and adds it to the result. Therefore, the implementation of `addFive` can reuse this function without spawning any further objects!

And now we come back to the question of why F# uses `FSharpFunc` instead of `Func`. The answer is… inheritance! Since an invocation of arguments involves not just a single function call but an entire chain, a really useful way of organizing this chain of invocations is by using good old-fashioned inheritance.

Composition

F# has special syntax for calling several functions one after another. In C#, if you need to take the value `x` and apply to it functions `g` and then `f`, you would simply write it as `f(g(x))`. In F#, the possibilities are more interesting.

Let us actually take a look at how these functions could be defined and used. We are going to consider the successive application of two functions, one that adds 5 to a number, the other being the one that doubles it.

```
let addFive x = x + 5
let timesTwo x = x * 2

printfn "%i" (addFive (timesTwo 3)) // 11
```

If you think about it, the number 3 above goes through a pipeline of operations: first it is fed to `timesTwo`, then to `addFive`. This notion of a pipeline is represented in code through the F# forward pipe and backward pipe operators, which can be used to implement the above operations as follow:

```
printfn "%i" (3 |> timesTwo |> addFive)
printfn "%i" (addFive <| (timesTwo <| 3))
```

Notice that while the forward operator `|>` example is very clean, the backward operator `<|` is much less so. The extra brackets are required due to associativity rules.

We might want to define a new function that applies `timesTwo` followed by `addFive` to any argument. Of course, you could simply define it as

```
let timesTwoAddFive x =
  x |> timesTwo |> addFive
```

However, F# also defines function composition operators >> (forward) and << (backward) for composing several functions into a single function. Naturally, their arguments must match.

```
let timesTwoAddFive = timesTwo >> addFive
printfn "%i" timesTwoAddFive 3 // 11
```

Functional-Related Language Features

While not central to the discussion of functional programming, certain features often go with it hand in hand. This includes the following:

- Tail recursion helps with defining algorithms in a recursive fashion.

- Discriminated unions allow very quick definitions of related types with primitive storage mechanics. Sadly, this feature breaks OCP because it's impossible to extend a discriminated union without changing its original definition.
- Pattern matching expands the domain of `if` statements with an ability to match against templates. This is omnipresent in F# (for lists, record types and others) and is now appearing in C#, too.
- Functional lists are a unique feature (entirely unrelated to `List<T>`), leveraging pattern matching and tail recursion.

These features are synergetic with the functional programming paradigm and can help the implementation of some of the patterns described in this book.

Creational Patterns

In a ‘managed’ language such as C#, the process of creating a new object is simple: just `new` it up and forget about it. Well, there’s `stackalloc`, but we’re mainly talking about objects that need to persist. Now, with the proliferation of Dependency Injection, another question is whether creating objects manually is still acceptable, or should we instead defer the creation of all key aspects of our infrastructure to specialized constructs such as Factories (more on them in just a moment!) or Inversion of Control containers?

Whichever option you choose, creation of objects can still be a chore, especially if the construction process is complicated or needs to abide by special rules. So that’s where creational patterns come in: they are common approaches related to the creation of objects.

Just in case you’re rusty on the ways an object can be constructed in C#, let’s recap the main approaches:

- Invocation of `new` creates an object on the managed heap. The object doesn’t need to be destroyed explicitly because the Garbage Collector (GC) will take care of it for us.
- Stack allocation with `stackalloc` allocates memory on the stack rather than the heap. Stack-allocated objects only exist in the scope they were created and get cleaned up auto when they go out of scope. This construct can only be used with value types.
- You can allocate unmanaged (native) memory with `Marshal.AllocHGlobal` and `CoTaskMemAlloc` and must explicitly free it with `Marshal.FreeHGlobal` and `CoTaskMemFree`. This is primarily needed for interoperation with unmanaged code.

Needless to say, some managed component might be working with unmanaged memory behind the scenes. This is one of the main reasons for the existence of the `IDisposable` interface. This interface has a single method, `Dispose()`, that can contain clean-up logic. If you are working with an object that implements `IDisposable`, it might make sense to wrap its use in a `using` statement (we now also have `using var`) so that its cleanup code gets executed as soon as the object is no longer needed.

Builder

The Builder pattern is concerned with the creation of *complicated* objects, i.e., objects that cannot be built up in a single-line constructor call. These types of objects may themselves be composed of other objects and might involve less-than-obvious logic, necessitating a separate component specifically dedicated to object construction.

I suppose it's worth noting beforehand that, while I said the Builder is concerned with *complicated* objects, we'll be taking a look at a rather trivial example. This is done purely for the purposes of space optimization, so that the complexity of the domain logic doesn't interfere with the reader's ability to appreciate the actual implementation of the pattern.

Scenario

Let's imagine that we are building a component that renders web pages. A page might consist of just a single paragraph (let's forget all the typical HTML trappings for now), and to generate it, you'd probably write something like the following:

```
var hello = "hello";
var sb = new StringBuilder();
sb.Append("<p>");
sb.Append(hello);
sb.Append("</p>");
WriteLine(sb);
```

This is some serious overengineering, Java-style, but it is a good illustration of one Builder that we've already got in the .NET Framework: the `StringBuilder`! `StringBuilder` is, of course, a separate component that is used for concatenating strings. It has utility methods such as `AppendLine()` so you can append both the text as well as a line break (as in `Enrivonment.NewLine`). But the real benefit to a `StringBuilder` is that, unlike string concatenation which results in lots of temporary strings, it just allocates a buffer and fills it up with text that is being appended.

So how about we try to output a simple unordered (bulleted) list with two items containg the words *hello* and *world*? A very simplistic implementation might look as follows:

```
var words = new[] { "hello", "world" };
sb.Append("<ul>");
foreach (var word in words)
{
  sb.AppendFormat("<li>{0}</li>", word);
}
sb.Append("</ul>");
WriteLine(sb);
```

This does in fact give us what we want, but the approach is not very flexible. How would we change this from a bulleted list to a numbered list? How can we add another item *after* the list has been created? Clearly, in this rigid scheme of ours, this is not possible once the `StringBuilder` has been initialized.

We might, therefore, go the OOP route and define an `HtmlElement` class to store information about each HTML tag:

```
class HtmlElement
{
  public string Name, Text;
  public List<HtmlElement> Elements = new List<HtmlElement>();
  private const int indentSize = 2;

  public HtmlElement() {}
  public HtmlElement(string name, string text)
  {
    Name = name;
    Text = text;
  }
}
```

This class models a single HTML tag which has a name and can also contain either text or a number of children, which are themselves `HtmlElements`. With this approach, we can now create our list in a more sensible fashion:

```
var words = new[] { "hello", "world" };
var tag = new HtmlElement("ul", null);
foreach (var word in words)
  tag.Elements.Add(new HtmlElement("li", word));
WriteLine(tag); // calls tag.ToString()
```

This works fine and gives us a more controllable, OOP-driven representation of a list of items. It also greatly simplifies other operations, such as the removal of entries. But the process of building up each `HtmlElement` is not very convenient, especially if that element has children or some special requirements. Consequently, we turn to the Builder pattern.

Simple Builder

The Builder pattern simply tries to outsource the piecewise construction of an object into a separate class. Our first attempt might yield something like this:

```
class HtmlBuilder
{
  protected readonly string rootName;
  protected HtmlElement root = new HtmlElement();

  public HtmlBuilder(string rootName)
  {
    this.rootName = rootName;
    root.Name = rootName;
  }

  public void AddChild(string childName, string childText)
  {
    var e = new HtmlElement(childName, childText);
    root.Elements.Add(e);
  }
  public override string ToString() => root.ToString();
}
```

This is a dedicated component for building up an HTML element. The constructor of the builder takes a `rootName`, which is the name of the root element that's being built: this can be `"ul"` if we are building an unordered list, `"p"` if we're making a paragraph, and so on. Internally, we store the root as an `HtmlElement`, and assign its `Name` in the constructor. But we also keep hold of the `rootName` so we can reset the builder later on if we wanted to.

The `AddChild()` method is the method that's intended to be used to add additional children to the current element, each child being specified as a name-text pair. It can be used as follows:

```
var builder = new HtmlBuilder("ul");
builder.AddChild("li", "hello");
builder.AddChild("li", "world");
WriteLine(builder.ToString());
```

You'll notice that, at the moment, the `AddChild()` method is `void`-returning. There are many things we could use the return value for, but one of the most common uses of the return value is to help us build a fluent interface.

Fluent Builder

Let's change our definition of `AddChild()` to the following:

```
public HtmlBuilder AddChild(string childName, string childText)
{
  var e = new HtmlElement(childName, childText);
  root.Elements.Add(e);
  return this;
}
```

By returning a reference to the builder itself, the builder calls can now be chained. This is what's called a *fluent interface*:

```
var builder = new HtmlBuilder("ul");
builder.AddChild("li", "hello").AddChild("li", "world");
WriteLine(builder.ToString());
```

The "one simple trick" of returning `this` allows you to build interfaces where several operations can be crammed into one statement. Note that `StringBuilder` itself also exposes a fluent interface. Fluent interfaces are generally nice, but making decorators that use them (e.g., using an automated tool such as ReSharper or Rider) can be a problem – we'll encounter this later.

Communicating Intent

We have a dedicated Builder implemented for an HTML element, but how will the users of our classes know how to use it? One idea is to simply *force* them to use the builder whenever they are constructing an object. Here's what you need to do:

```
class HtmlElement
{
  protected string Name, Text;
  protected List<HtmlElement> Elements = new List<HtmlElement>();
  protected const int indentSize = 2;

  // hide the constructors!
  protected HtmlElement() {}
  protected HtmlElement(string name, string text)
  {
    Name = name;
    Text = text;
  }

  // factory method
  public static HtmlBuilder Create(string name) => new HtmlBuilder(name);
}
```

Our approach is two-pronged. First, we have hidden all constructors, so they are no longer available. We have also hidden the implementation details of the Builder itself, something we haven't done previously. We have, however, created a Factory Method (this is a design pattern we shall discuss later) for creating a builder right out of the `HtmlElement`. And it's a static method, too! Here's how one would go about using it:

```
var builder = HtmlElement.Create("ul");
builder.AddChild("li", "hello")
  .AddChild("li", "world");
WriteLine(builder);
```

In the example above, we are *forcing* the client to use the static `Create()` method because, well, there's really no other way to construct an `HtmlElement` – after all, all the constructors are `protected`. So the client creates an `HtmlBuilder` and is then forced to interact with it in the construction of an object. The last line of the listing simply prints the object being constructed.

But let's not forget that our ultimate goal is to build an `HtmlElement`, and so far we have no way of getting to it! So the icing on the cake can be an implementation of `implicit operator HtmlElement` on the builder to yield the final value:

```
protected HtmlElement root = new HtmlElement();

public static implicit operator HtmlElement(HtmlBuilder builder)
{
  return builder.root;
}
```

The addition of the operator allows us to write the following:

```
HtmlElement root = HtmlElement
  .Create("ul")
  .AddChildFluent("li", "hello")
  .AddChildFluent("li", "world");
WriteLine(root);
```

Regrettably, there is no way of explicitly telling other users to use the API in this manner. Hopefully the restriction on constructors coupled with the presence of the static `Create()` method encourages the user to use the builder, but, in addition to the operator, it might make sense to also add a corresponding `Build()` function to `HtmlBuilder` itself:

```
public HtmlElement Build() => root;
```

Composite Builder

Let us continue the discussion of the Builder pattern with an example where multiple builders are used to build up a single object. This scenario is relevant to situations where the building process is so complicated that the builder itself becomes subject to the Single Responsibility Principle and needs to be framented into smaller parts.

Let's say we decide to record some information about a person:

```
public class Person
{
  // address
  public string StreetAddress, Postcode, City;

  // employment info
  public string CompanyName, Position;
  public int AnnualIncome;
}
```

There are two aspects to `Person`: their address and employment information. What if we want to have separate builders for each – how can we provide the most convenient API? To do this, we'll construct a composite builder. This construction is not trivial, so pay attention: even though we want two separate builders for job and address information, we'll spawn no fewer than *three* distinct classes.

We'll call the first class `PersonBuilder`:

```
public class PersonBuilder
{
  // the object we're going to build
  protected Person person; // this is a reference!

  public PersonBuilder() => person = new Person();
  protected PersonBuilder(Person person) => this.person = person;

  public PersonAddressBuilder Lives => new PersonAddressBuilder(person);
  public PersonJobBuilder Works => new PersonJobBuilder(person);

  public static implicit operator Person(PersonBuilder pb)
  {
    return pb.person;
  }
}
```

This is *much* more complicated than our simple Builder earlier, so let’s discuss each member in turn:

- The reference `person` is a reference to the object that’s being built. This field is marked `protected`, and this is done deliberately for the sub-builders. It’s worth noting that this approach only works for reference types - if `person` were a `struct`, we would encounter unnecessary duplication.
- `Lives` and `Works` are properties returning builder facets: those sub-builders that initialize the address and employment information separately.
- `operator Person` is a trick that we’ve used before.

One very important point to note is the constructors: instead of just initializing the `person` reference with a `new Person()` everywhere, we only do so in the public, parameterless constructor. There is another constructor that takes a reference and saves it – this constructor is designed to be used by inheritors and not by the client, that’s why it is protected. The reason why things are set up this way is so that a `Person` is instantiated only once per use of the builder, even if the sub-builders are used.

Now, let’s take a look at the implementation of a sub-builder class:

```
public class PersonAddressBuilder : PersonBuilder
{
  public PersonAddressBuilder(Person person) : base(person)
  {
    this.person = person;
  }

  public PersonAddressBuilder At(string streetAddress)
  {
    person.StreetAddress = streetAddress;
    return this;
  }

  public PersonAddressBuilder WithPostcode(string postcode)
  {
    person.Postcode = postcode;
    return this;
  }

  public PersonAddressBuilder In(string city)
  {
    person.City = city;
    return this;
  }
};
```

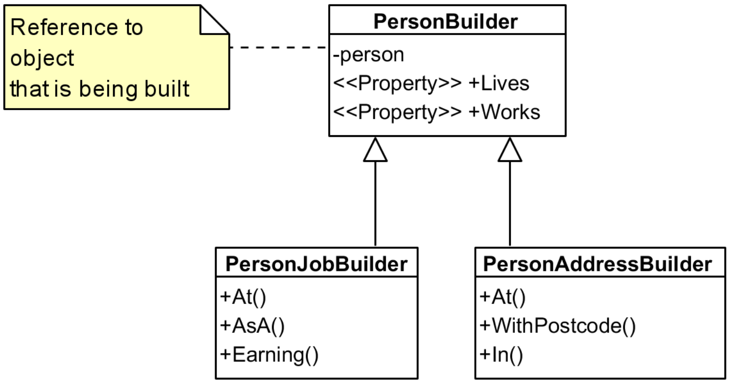
As you can see, `PersonAddressBuilder` provides a fluent interface for building up a person's address. Note that it actually *inherits* from `PersonBuilder` (meaning it has acquired the `Lives` and `Works` properties). It has a constructor that takes and stores a reference to the object that's being constructed, so when you use these sub-builders, you are always working with just a single instance of `Person` - you are not accidentally spawning multiple instances. It is *critical* that the base constructor is called – if it is not, the sub-builder will call the parameterless constructor automatically, causing the unnecessary instantiation of additional `Person` instances.

As you can guess, `PersonJobBuilder` is implemented in identical fashion, so I'll omit it here.

And now, the moment you've been waiting for – an example of these builders in action:

```
var pb = new PersonBuilder();
Person person = pb
  .Lives
    .At("123 London Road")
    .In("London")
    .WithPostcode("SW12BC")
  .Works
    .At("Fabrikam")
    .AsA("Engineer")
    .Earning(123000);

WriteLine(person);
// StreetAddress: 123 London Road, Postcode: SW12BC, City: London,
// CompanyName: Fabrikam, Position: Engineer, AnnualIncome: 123000
```

Can you see what's happening here? We make a builder, and then use the `Lives` property to get us a `PersonAddressBuilder` but once we're done initializing the address information, we simply call `Works` and switch to using a `PersonJobBuilder` instead. And just in case you need a visual illustration of what we just did, it's rather uncomplicated:



When we're done with the building process, we use the same implicit conversion trick as before to get the object being built-up as a `Person`. Alternatively, you can invoke `Build()` to get the same result.

There's one fairly obvious downside to this approach: it's not extensible. Generally speaking, it's a bad idea for a base class to be aware of its own subclasses, yet this is precisely what's happening here – `PersonBuilder` is aware of its own children by exposing them through special APIs. If you wanted to have an additional sub-builder (say, a `PersonEarningsBuilder`), you would have to break OCP and edit `PersonBuilder` directly; you cannot simply subclass it to add an interface member.

Builder Parameter

As I have demonstrated, the only way to coerce the client to use a builder rather than constructing the object directly is to make the object's constructors inaccessible. There are situations, however, when you want to explicitly force the user to interact with the builder from the outset, possibly concealing even the object they're actually building.

For example, suppose you have an API for sending emails, where each email is described internally like this:

```
public class Email
{
  public string From, To, Subject, Body;
  // other members here
}
```

Note that I said *internally* here – you have no desire to let the user interact with this class directly, perhaps because there is some additional service information stored in it. Keeping it public is fine though, provided you expose no API that allows the client to send an `Email` directly. Some parts of the email (for example, the `Subject`) are optional, so the object doesn't have to be fully specified.

You decide to implement a fluent builder that people will use for constructing an `Email` behind the scenes. It may appear as follows:

```
public class EmailBuilder
{
  private readonly Email email;
  public EmailBuilder(Email email) => this.email = email;

  public EmailBuilder From(string from)
  {
    email.From = from;
    return this;
  }

  // other fluent members here
}
```

Now, to coerce the client to use only the builder for sending emails, you can implement a `MailService` as follows:

```
public class MailService
{
  public class EmailBuilder { ... }

  private void SendEmailInternal(Email email) {}

  public void SendEmail(Action<EmailBuilder> builder)
  {
    var email = new Email();
    builder(new EmailBuilder(email));
    SendEmailInternal(email);
  }
}
```

As you can see, the `SendEmail()` method that clients are meant to use takes a function, not just a set of parameters or a prepackaged object. This function takes an `EmailBuilder` and then is expected to use the builder to construct the body of the message. Once that is done, we use the internal mechanics of `MailService` to process a fully initialized `Email`.

You'll notice there's a clever bit of subterfuge here: instead of storing a reference to an email internally, the builder gets that reference in the constructor argument.

The reason why we implement it this way is so that `EmailBuilder` wouldn't have to expose an `Email` publicly anywhere in its API.

Here's what the use of this API looks like from the client's perspective:

```
var ms = new MailService();
ms.SendEmail(email => email.From("foo@bar.com")
                           .To("bar@baz.com")
                           .Body("Hello, how are you?"));
```

Long story short, the Builder Parameter approach forces the consumer of your API to use a builder, whether they like it or not. This `Action` trick that we employ ensures that the client has a way of receiving an already-initialized builder object.

Builder Extension with Recursive Generics

One interesting problem that doesn't just affect the fluent Builder but *any* class with a fluent interface is the problem of inheritance. Is it possible (and realistic) for a fluent builder to inherit from another fluent builder? It is, but it's not easy.

Here is the problem. Suppose you start out with the following (very trivial) object that you want to build up:

```
public class Person
{
  public string Name;
  public string Position;
}
```

You make a base class Builder that facilitates the construction of `Person` objects:

```
public abstract class PersonBuilder
{
  protected Person person = new Person();
  public Person Build()
  {
    return person;
  }
}
```

Followed by a dedicated class for specifying the `Person`'s name:

```
public class PersonInfoBuilder : PersonBuilder
{
  public PersonInfoBuilder Called(string name)
  {
    person.Name = name;
    return this;
  }
}
```

This works, and there is absolutely no issue with it. But now, suppose we decide to subclass `PersonInfoBuilder` so as to also specify employment information. You might write something like this:

```
public class PersonJobBuilder : PersonInfoBuilder
{
  public PersonJobBuilder WorksAsA(string position)
  {
    person.Position = position;
    return this;
  }
}
```

Sadly, we've now broken the fluent interface and rendered the entire set-up unusable:

```
var me = Person.New
  .Called("Dmitri")  // returns PersonInfoBuilder
  .WorksAsA("Quant") // will not compile
  .Build();
```

Why won't the above compile? It's simple: `Called()` returns `this`, which is an object of type `PersonInfoBuilder`; that object simply doesn't have the `WorksAsA()` method!

You might think the situation is hopeless, but it's not: you can design your fluent APIs with inheritance in mind, but it's going to be a bit tricky. Let's take a look at what's involved by redesigning the `PersonInfoBuilder` class. Here is its new incarnation:

```
public class PersonInfoBuilder<SELF> : PersonBuilder
  where SELF : PersonInfoBuilder<SELF>
{
  public SELF Called(string name)
  {
    person.Name = name;
    return (SELF) this;
  }
}
```

If you're not familiar with recursive generics, the code above might seem rather overwhelming, so let's discuss what we actually did and why.

Firstly, we essentially introduced a new generic argument, `SELF`. What's more curious is that this `SELF` is specified to be an inheritor of `PersonInfoBuilder<SELF>`; in other words, the generic argument of the class is required to inherit from this exact class. This may seem like madness, but is actually a very popular trick for doing CRTP-style inheritance in C#[8]. Essentially, we are enforcing an inheritance chain: we are saying that `Foo<Bar>` is only an acceptable specialization if `Foo` derives from `Bar`, and all other cases should fail the `where` constraint.

The biggest problem in fluent interface inheritance is being able to return a `this` reference that is typed to the class you're currently in, even if you are calling a

[8]CRTP stands for Curiously Recurring Template Pattern; it is a popular C++ pattern which looks like this: `class Foo<T> : T`. In other words, you inherit from a generic parameter, something that's impossible in C#.

fluent interface member of a *base* class. The only way to efficiently propagate this is by having a generic parameter (the SELF) that permeates the entire inheritance hierarchy.

To appreciate this, we need to look at PersonJobBuilder, too:

```
public class PersonJobBuilder<SELF>
  : PersonInfoBuilder<PersonJobBuilder<SELF>>
  where SELF : PersonJobBuilder<SELF>
{
  public SELF WorksAsA(string position)
  {
    person.Position = position;
    return (SELF) this;
  }
}
```

Look at its base class! It's not just an ordinary PersonInfoBuilder as before, instead it's a PersonInfoBuilder<PersonJobBuilder<SELF>>! So when we inherit from a PersonInfoBuilder, we set its SELF to PersonJobBuilder<SELF> so that all of its fluent interfaces return the correct type, *not* just the type of the owning class.

Does this make sense? If not, take your time and look through the source code once again. Here, let's test your understanding: suppose I introduce another member called DateOfBirth and a corresponding PersonDateOfBirthBuilder, what class would it inherit from?

If you answered

```
PersonInfoBuilder<PersonJobBuilder<PersonBirthDateBuilder<SELF>>>
```

then you are wrong, but I cannot blame you for trying. Think about it: PersonJob-Builder is *already* a PersonInfoBuilder, so that information doesn't need to be restated explicitly as part of the inheritance type list. Instead, you would define the builder as follows:

```
public class PersonBirthDateBuilder<SELF>
  : PersonJobBuilder<PersonBirthDateBuilder<SELF>>
  where SELF : PersonBirthDateBuilder<SELF>
{
  public SELF Born(DateTime dateOfBirth)
  {
    person.DateOfBirth = dateOfBirth;
    return (SELF)this;
  }
}
```

The final question we have is this: how do we actually construct such a builder, considering that it *always* takes a generic argument? Well, I'm afraid you now need a new *type*, not just a variable. So, for example, the implementation of `Person.New` (the property that starts off the construction process) can be implemented as follows:

```
public class Person
{
  public class Builder : PersonJobBuilder<Builder>
  {
    internal Builder() {}
  }

  public static Builder New => new Builder();

  // other members omitted
}
```

This is probably the most annoying implementation detail: the fact that you need to have a non-generic inheritor of a recursive generic type in order to use it.

That said, putting everything together, you can now use the builder, leveraging all methods in the inheritance chain:

```
var builder = Person.New
  .Called("Natasha")
  .WorksAsA("Doctor")
  .Born(new DateTime(1981, 1, 1));
```

Lazy Functional Builder

The previous example of using recursive generics requires a lot of work. A fair question to ask is: should inheritance have been used to extend the builders? After all, we could have used extension methods instead.

If we adopt a functional approach, the implementation becomes a lot simpler, without the need for recursive generics. Let's once again build up a `Person` class defined as follows:

```
public class Person
{
  public string Name, Position;
}
```

This time round, we'll define a *lazy* builder that only constructs the object when its `Build()` method is called. Until that time, it will simply keep a list of `Action`s that need to be performed when an object is built:

```
public sealed class PersonBuilder
{
  private readonly List<Func<Person, Person>> actions =
    new List<Func<Person, Person>>();

  public PersonBuilder Do(Action<Person> action)
    => AddAction(action);

  public Person Build()
    => actions.Aggregate(new Person(), (p, f) => f(p));

  private PersonBuilder AddAction(Action<Person> action)
  {
    actions.Add(p => { action(p); return p; });
    return this;
  }
}
```

The idea is simple: instead of having a mutable ‘object under construction’ that is modified as soon as any builder method is invoked, we simply store a list of actions that need to be applied upon the object whenever someone calls `Build()`. But there are additional complications in our implementation.

The first is that the action taken upon the person, while take as an `Action<T>` parameter is actually stored as a `Func<T,T>`. The motivation behind this is that providing this fluent interface, we’re allowing for the `Aggregate()` call inside `Build()` to work correctly. Of course, we could have used a good old-fashioned `ForEach()` instead.

The second complication is that, in order to allow OCP-conformant extensibility, we really don’t want to expose `actions` as a public member, since this would allow far too many operations (e.g., arbitrary removal) on the list that we don’t necessarily want expose to whoever extends this builder in the future. Instead, we publicly expose only a single operation, `Do()`, that allows you to specify an action to be performed on the object under construction. That action is then added to the overall set of actions.

Under this paradigm, we can now give this builder a concrete method for specifying a `Person`’s name:

```
public PersonBuilder Called(string name)
  => Do(p => p.Name = name);
```

But now, thanks to the way the builder is structured, we can use extension methods instead of inheritance to give the builder additional functionality, such as an ability to specify a person’s position:

```
public static class PersonBuilderExtensions
{
  public static PersonBuilder WorksAs
    (this PersonBuilder builder, string position)
    => builder.Do(p => p.Position = position);
}
```

With this approach, there are no inheritance issues and no recursive magic. Any time we want additional behaviors, we simply add them as extension methods, preserving adherence to the OCP.

And here is how you would use this set-up:

```
var person = new PersonBuilder()
  .Called("Dmitri")
  .WorksAs("Programmer")
  .Build();
```

Strictly speaking, the functional approach above can be made into a reusable generic base class that can be reused for building different objects. The only issue is that you'll have to propagate the derived type into the base class which, once again, requires recursive generics.

You would define the base `FunctionalBuilder` as:

```
public abstract class FunctionalBuilder<TSubject, TSelf>
  where TSelf: FunctionalBuilder<TSubject, TSelf>
  where TSubject : new()
{
  private readonly List<Func<TSubject, TSubject>> actions
    = new List<Func<TSubject, TSubject>>();


  public TSelf Do(Action<TSubject> action)
    => AddAction(action);

  private TSelf AddAction(Action<TSubject> action)
  {
    actions.Add(p => {
      action(p);
      return p;
    });
    return (TSelf) this;
  }

  public TSubject Build()
    => actions.Aggregate(new TSubject(), (p, f) => f(p));
}
```

With `PersonBuilder` now simplifying to:

```
public sealed class PersonBuilder
  : FunctionalBuilder<Person, PersonBuilder>
{
  public PersonBuilder Called(string name)
    => Do(p => p.Name = name);
}
```

and the `PersonBuilderExtensions` class remaining as it was. With this approach, you could easily reuse `FunctionalBuilder` as a base class for other functional builders in your application. Notice that, under the functional paradigm, we're still sticking to the idea that the derived builders are all `sealed` and extended through the use of extension methods.

DSL Construction in F#

Many programming languages (such as Groovy, Kotlin or F#) try to throw in a language feature that will simplify the process of creating DSLs — Domain-Specific Languages, i.e. small languages that help describe a particular problem domain. Many applications of such embedded DSLs are used to implement the Builder pattern. For example, if you want to build an HMTL page, you don't have to fiddle with classes and methods directly; instead, you can write something which very much approaches HTML, right in your code!

The way this is made possible in F# is using list comprehensions: the ability to define lists without any explicit calls to builder methods. For example, if you wanted to support HTML paragraphs and images, you could define the following builder functions:

```
let p args =
  let allArgs = args |> String.concat "\n"
  ["<p>"; allArgs; "</p>"] |> String.concat "\n"

let img url = "<img src=\"" + url + "\"/>"
```

Notice that whereas the `img` tag only has a single textual parameter, the <p> tag accepts a sequence of `args`, allowing it to contain any number of inner HTML elements, including ordinary plain text. We could therefore construct a paragraph containing both text and an image:

```
let html =
  p [
    "Check out this picture";
    img "pokemon.com/pikachu.png"
  ]
printfn "%s" html
```

This results in the following output:

```
<p>
Check out this picture
<img src="pokemon.com/pikachu.png"/>
</p>
```

This approach is used in web frameworks such as WebSharper. There are many variations to this approach, including the use of record types (letting people use curly braces instead of lists), custom operators for specifying plain text, and more.[9]

It's important to note that this approach is only convenient when we are working with an immutable, append-only structure. Once you start dealing with mutable objects (e.g., using a DSL to construct a definition for a Microsoft Project document), you end up falling back into OOP. Sure, the end-result DSL syntax is still very convenient to use, but the plumbing required to make it work is anything but pretty.

[9]For an example, see Tomas Petricek's snippet for an F#-based HTML-constructing DSL at http://fssnip.net/hf.

Summary

The goal of the Builder pattern is to define a component dedicated entirely to piecewise construction of a complicated object or set of objects. We have observed the following key characteristics of a Builder:

- Builders can have a fluent interface that is usable for complicated construction using a single invocation chain. To support this, builder functions should return `this`.
- To force the user of the API to use a Builder, we can make the target class constructors inaccessible and then define a static `Create()` function that returns an instance of the builder. (The naming is up to you, you can call it `Make()`, `New()` or something else.)
- A builder can be coerced to the object itself by defining the appropriate implicit conversion operator.
- You can force the client to use a builder by specifying it as part of a parameter function. This way you can hide the object that's built built entirely.
- A single builder interface can expose multiple sub-builders. Through clever use of inheritance and fluent interfaces, one can jump from one builder to another with ease.
- Inheritance of fluent interfaces (not just for builders) is possible through recursive generics.

Just to re-iterate something that I've already mentioned, the use of the Builder pattern makes sense when the construction of the object is a *non-trivial* process. Simple objects that are unambiguously constructed from a limited number of sensibly named constructor parameters should probably use a constructor (or dependency injection) without necessitating a Builder as such.

Factories

> I had a problem and tried to use Java, now I have a ProblemFactory. – *Old Java joke.*

This chapter covers two GoF patterns: *Factory Method* and *Abstract Factory*. These patterns are closely related, so we'll discuss them together. The truth, though, is that the real design pattern is called *Factory* and that both Factory Method and Abstract Factory are simply variations that are important, but certainly not as important as the main thing.

Scenario

Let's begin with a motivating example. Suppose you want to store information about a `Point` in Cartesian (X-Y) space. So you go ahead and implement something like this:

```
public class Point
{
  private double x, y;

  public Point(double x, double y)
  {
    this.x = x;
    this.y = y;
  }
}
```

So far, so good. But now, you also want to initialize the point from *polar* coordinates instead. You need another constructor with the signature:

```
Point(float r, float theta)
{
  x = r * Math.Cos(theta);
  y = r * Math.Sin(theta);
}
```

Unfortunately, you've already got a constructor with two `floats`, so you cannot have another one.[10] What do you do? One approach is to introduce an enumeration:

```
public enum CoordinateSystem
{
  Cartesian,
  Polar
}
```

And then add another parameter to the point constructor:

```
public Point(double a,
  double b, // names do not communicate intent
  CoordinateSystem cs = CoordinateSystem.Cartesian)
{
  switch (cs)
  {
    case CoordinateSystem.Polar:
      x = a * Math.Cos(b);
      y = a * Math.Sin(b);
      break;
    default:
      x = a;
      y = b;
      break;
  }
}
```

[10]Some programming languages, most notably Objective-C and Swift, do allow overloading of functions that only differ by parameter names. Unfortunately, this idea results in a viral propagation of parameter names in all calls. I still prefer positional parameters, most of the time.

Notice how the names of the first two arguments were changed to `a` and `b`: we can no longer afford telling the user which co-ordinate system those values should come from. This is a clear loss of expressivity when compared with using `x`, `y`, `rho` and `theta` to communicate intent.

Overall, our constructor design is usable, but ugly. In particular, in order to add some third co-ordinate system, for example, you would need to:

- Give `CoordinateSystem` a new enumeration value.
- Change the constructor to support the new co-ordinate system.

There must be a better way of doing this.

Factory Method

The trouble with the constructor is that its name always matches the type. This means we cannot communicate any extra information in it, unlike in an ordinary method. Also, given the name is always the same, we cannot have two overloads one taking `x,y` and another taking `r,theta`.

So what can we do? Well, how about making the constructor `protected`[11] and then exposing some static functions for creating new points?

```
public class Point
{
  protected Point(double x, double y)
  {
    this.x = x;
    this.y = y;
  }

  public static Point NewCartesianPoint(double x, double y)
  {
    return new Point(x, y);
  }
```

[11]Whenever you want to prevent a client from accessing something, I always recommend you make it `protected` rather than `private` because then you make the class inheritance-friendly.

```
  public static Point NewPolarPoint(double rho, double theta)
  {
    return new Point(rho*Math.Cos(theta), rho*Math.Sin(theta));
  }

  // other members omitted
}
```

Each of the above static functions is called a Factory Method. All it does is create a `Point` and return it, the advantages being that both the name of the method as well as the names of the arguments clearly communicate what kind of co-ordinates are required.

Now, to create a point, you simply write

```
var point = Point.NewPolarPoint(5, Math.PI / 4);
```

From the above, we can clearly surmise that we are creating a new point with polar co-ordinates $r = 5$ and $theta = \pi/4$.

Asynchronous Factory Method

When we talk about constructors, we always assume that the body of the constructor is synchronous. A constructor always returns the type of the constructed object – it cannot return a `Task` or `Task<T>`, therefore it cannot be asynchonous. And yet, there are situations where you do want the object to be initialized in an asynchronous fashion.

There are (at least) two ways this can be handled. The first is *by convention*: we simply agree that any asynchronously initialized type has a method called, say, `InitAsync()`:

```
public class Foo
{
  private async Task InitAsync()
  {
    await Task.Delay(1000);
  }
}
```

The assumption here is that the client would recognize this member and will remember to call it, as in:

```
var foo = new Foo();
await foo.InitAsync();
```

But this is very optimistic. A better approach is to hide the constructor (make it `protected`) and then create a `static` factory method that both creates an instance of `Foo` and initializes it. We can even give it a fluent interface so that the resulting object is ready to use:

```
public class Foo
{
  protected Foo() { /* init here */ }

  public static Task<Foo> CreateAsync()
  {
    var result = new Foo();
    return result.InitAsync();
  }
}
```

This can now be used as:

```
var foo = await Foo.CreateAsync();
```

Naturally, if you need constructor arguments, you can add them to the constructor and forward them from the factory method.

Factory

Just like with Builder, we can take all the Point-creating functions out of Point into a separate class, that we call a Factory. It's actually very simple:

```
class PointFactory
{
  public static Point NewCartesianPoint(float x, float y)
  {
    return new Point(x, y); // needs to be public
  }
  // same for NewPolarPoint
}
```

It's worth noting that the Point constructor can no longer be private or protected because it needs to be externally accessible. Unlike C++, there is no friend keyword for us to use; we'll resort to a different trick later on.

But for now, that's it – we have a dedicated class specifically designed for creating Point instances, to be used as follows:

```
var myPoint = PointFactory.NewCartesian(3, 4);
```

Inner Factory

An inner factory is simply a factory that is an inner (nested) class within the type it creates. The reason why inner factories exist is because inner classes can access the outer class' private members and, conversely, an outer class can access an inner class' private members. This means that our Point class can also be defined as follows:

```
public class Point
{
  // typical members here

  // note the constructor is again private
  private Point(double x, double y) { ... }

  public static class Factory
  {
    public static Point NewCartesianPoint(double x, double y)
    {
      return new Point(x, y); // using a private constructor
    }
    // similar for NewPolarPoint()
  }
}
```

Okay, so what's going on here? Well, we've stuck the factory right into the class the factory creates. This is convenient if a factory only works with one single type, and not so convenient if a factory relies on several types (and pretty much impossible if it needs their `private` members, too).

With this approach, we can now write

```
var point = Point.Factory.NewCartesianPoint(2, 3);
```

You might find this approach familiar because several parts of the .NET framework use this approach to expose factories. For example, the TPL lets you spin up new tasks with `Task.Factory.StartNew()`.

Physical Separation

If you don't like the idea of having the entire definition of the `Factory` being placed into your `Point.cs` file, you can use the `partial` keyword because, guess what, it works on inner classes too. First, in `Point.cs`, you would modify the `Point` type to now read

```
public partial class Point { ... }
```

Then, simply make a new file (e.g., `Point.Factory.cs`) and, inside it, define another part of `Point`, i.e.,

```
public partial class Point
{
  public static class Factory
  {
    // as before
  }
}
```

That's it! You've now physically separated the factory from the type itself, even though logically they are still entwined since one contains the other.

Abstract Factory

So far, we've been looking at the construction of a single object. Sometimes, you might be involved in the creation of families of objects. This is actually a pretty *rare* case, so unlike Factory Method and the plain old Factory pattern, Abstract Factory is a pattern that only shows up in complicated systems. We need to talk about it, regardless, primarily for historical reasons.

The scenario we're going to take a look at here is a scenario that is show by many sources all over the web, so I hope you'll forgive the repetition. We'll consider a hierarchy of geometrical shapes that we want to draw. We'll consider only shapes that have lines joining at right angles:

```
public interface IShape
{
  void Draw();
}

public class Square : IShape
{
  public void Draw() => Console.WriteLine("Basic square");
}

public class Rectangle : IShape
{
  public void Draw() => Console.WriteLine("Basic rectangle");
}
```

Both `Square` and `Rectangle`, which implement the `IShape` interface, form a kind of family: they are simple geometric shapes drawn using straight lines connected at right angles. We can now imagine another parallel reality where right angles are considered aesthetically displasing, where both squares and rectangles would have their corners rounded:

```
public class RoundedSquare : IShape
{
  public void Draw() => Console.WriteLine("Rounded square");
}

public class RoundedRectangle : IShape
{
  public void Draw() => Console.WriteLine("Rounded rectangle");
}
```

You'll notice that the two hierarchies are related conceptually, but there's no code element to indicate that they are part of the same thing. We could introduce such an element in a number of ways, one being a simple enumeration with all the possible shapes that the system supports:

```
public enum Shape
{
  Square,
  Rectangle
}
```

So we now have two *families* of objects: a family of basic shapes and a family of rounded shapes. With that in mind, we can create a factory for basic shapes:

```
public class BasicShapeFactory : ShapeFactory
{
  public override IShape Create(Shape shape)
  {
    switch (shape)
    {
      case Shape.Square:
        return new Square();
      case Shape.Rectangle:
        return new Rectangle();
      default:
        throw new ArgumentOutOfRangeException(
          nameof(shape), shape, null);
    }
  }
}
```

And a similar RoundedShapeFactory for rounded shapes. Since the methods of these two factories would be identical, they can both inherit from an Abstract Factory defined as follows:

```
public abstract class ShapeFactory
{
  public abstract IShape Create(Shape shape);
}
```

What we've ended up with is a situation where a hierarchy of shapes got a corresponding hierarchy of factories. We can now create a method that will yield a particular type of factory on the basis of whether shape rounding is actually required:

```
public static ShapeFactory GetFactory(bool rounded)
{
  if (rounded)
    return new RoundedShapeFactory();
  else
    return new BasicShapeFactory();
}
```

And that's it! We now have a configurable way of instantiating not just individual objects, but entire families of objects:

```
var basic = GetFactory(false);
var basicRectangle = basic.Create(Shape.Rectangle);
basicRectangle.Draw(); // Basic rectangle

var roundedSquare = GetFactory(true).Create(Shape.Square);
roundedSquare.Draw(); // Rounded square
```

Natually, the kind of manual configuration that we've done above could easily be done using an IoC container – you simply define whether requests for a `ShapeFactory` should yield instances of `BasicShapeFactory`, `RoundedShapeFactory` or some other factory type. In fact, unlike the `GetFactory()` method above, the use of an IoC container will not suffer from a (trivial) OCP violation since no code other than the container configuration would have to be rewritten if a new `ShapeFactory` were to be introduced.

There's an additional thing that has to be said about the relationship between the `Shape` enum and the `IShape` inheritors. Strictly speaking, though our example works, there's no real enforcement that the enum members correspond 1-to-1 to the entire set of possible hierarchies. You could introduce such validations at compile-time, but to derive the set of enum members (via T4/Roslyn, perhaps?) you would probably have to introduce additional `IShape`-implementing abstract classes (e.g., `BasicShape` and `RoundedShape`) so you have clear delineation between the two different hierarchies. It's up to you to decide whether or not this approach makes sense in your particular case.

Delegate Factories in IoC

One problem that we encounter when we work with Dependency Injection and IoC containers is that, sometimes, you have an object which has a bunch of services it depends on (that can be injected) but it also has some constructor arguments that you need.

For example, given a service such as

```
public class Service
{
  public string DoSomething(int value)
  {
    return $"I have {value}";
  }
}
```

imagine a domain object that depends on this service, but also has a constructor argument that needs to be provided and is subsequently used in the dependent service:

```
public class DomainObject
{
  private Service service;
  private int value;

  public DomainObject(Service service, int value)
  {
    this.service = service;
    this.value = value;
  }

  public override string ToString()
  {
    return service.DoSomething(value);
  }
}
```

How would you configure your DI container (e.g., Autofac) to construct an instance of `DomainObject` that injects the service and also specifies the value of 42 for the value? Well, there's a brute force approach, but it's rather ugly:

```
var cb = new ContainerBuilder();
cb.RegisterType<Service>();
cb.RegisterType<DomainObject>();

using var container = cb.Build();
var dobj = container.Resolve<DomainObject>(
  new PositionalParameter(1, 42));
Console.WriteLine(dobj); // I have 42
```

This works, but this code is brittle and not refactoring-friendly. What if the position of the parameter `value` changes? This would invalidate the `Resolve()` step. And yes, we can try getting the parameter by name, but then again, the ability to refactor (e.g., rename) the constructor will suffer.

Luckily there is a solution for this problem and it's called a Delegate Factory. A delegate factory is, quite simply, a delegate that initializes an object, but it only requires you to pass the parameters that are not injected automatically. For example, a delegate factory for our domain object is as simple as:

```
public class DomainObject
{
  public delegate DomainObject Factory(int value);
  // other members here
}
```

Now, when you use the `DomainObject` in your IoC container, instead of resolving the object itself, you resolve the factory!

```
var factory = container.Resolve<DomainObject.Factory>();
var dobj2 = factory(42);
Console.WriteLine(dobj2); // I have 42
```

The registration steps remain exactly the same. What happens behind the scenes is this: the IoC container initializes the delegate to construct an instance of an object that makes use of both the dependent services and the values provided in the delegate. Then, when you resolve it, the delegate is fully initialized and is ready to use!

Functional Factory

Under the purely functional paradigm, the Factory pattern is of limited use, since F# prefers to work with concrete types whenever possible, using functions and functional composition to express variability in implementation.

If you wanted to go with interfaces (which F# allows), then, given the following definition

```
type ICountryInfo =
  abstract member Capital : string

type Country =
  | USA
  | UK
```

you could define a factory function that, for a given country, yields a properly initialized `ICountryInfo` object:

```
let make country =
  match country with
  | USA -> { new ICountryInfo with
             member x.Capital = "Washington" }
  | UK -> { new ICountryInfo with
            member x.Capital = "London" }
```

Suppose you want to be able to create a country by specifying its name as a string. In this case, in addition to having a freestanding function that gives you the right `Country` type, you can have a static factory method very similar to the ones we have in the OOP world:

```
type Country =
  | USA
  | UK
with
  static member Create = function
    | "USA" | "America" -> USA
    | "UK" | "England" -> UK
    | _ -> failwith "No such country"

let usa = Country.Create "America"
```

Naturally, the Abstract Factory approach is similarly implementable using functional composition instead of inheritance.

Summary

Let's recap the terminology:

- A *factory method* is a class member that acts as a way of creating object. It typically replaces a constructor.
- A *factory* is typically a separate class that knows how to construct objects, though if you pass a function (as in `Func<T>` or similar) that constructs objects, this argument is also called a factory.
- An *abstract factory* is, as its name suggests, an abstract class that can be inherited by concrete classes that offer a family of types. Abstract factories are rare in the wild.

A factory has several critical advantages over a constructor call, namely:

- A factory can say *no*, meaning that instead of actually returning an object it can return, for example, a `null` or `None` of some `Option<T>` type.
- Naming is better and unconstrained, unlike constructor name.
- A single factory can make objects of many different types.
- A factory can exhibit polymorphic behavior, instantiating a class and returning it through a reference to its base class or interface.

- A factory can implement caching and other storage optimizations; it is also a natural choice for approaches such as pooling or the Singleton pattern.
- A factory can change its behavior at runtime; `new` is expected to always yield a new instance.

Factory is different from Builder in that, with a Factory, you typically create an object in one go (i.e., a single statement), whereas with Builder, you construct the object piecewise – either through several statements or, possibly, in a single statement if the builder supports a fluent interface.

Prototype

Think about something you use every day, like a car or a mobile phone. Chances are, it wasn't designed from scratch; instead, the manufacturer chose an *existing* design, made some improvements, made it visually distintive from the old design (so people could show off) and started selling it, retiring the old product. It's a natural state of affairs, and in the software world, we get a similar situation: sometimes, instead of creating an entire object from scratch (the Factory and Builder patterns can help here), you want to take a preconstructed object and either use a copy of it (which is easy) or, alternatively, customize it a little.

And this leads us to the idea of having a Prototype: a model object that we can make copies of, customize those copies and then use them. The challenge of the Prototype pattern is really the copying part; everything else is easy.

Deep vs. Shallow Copying

Suppose we define a class `Person` as

```
public class Person
{
  public readonly string Name;
  public readonly Address Address;

  public Person(string name, Address address) { ... }
}
```

with the `Address` defined as

```
public class Address
{
  public readonly string StreetName;
  public int HouseNumber;

  public Address(string streetName, int houseNumber) { ... }
}
```

Suppose John Smith and Jane Smith are neighbors. It should be possible to construct John, then just copy him and change the house number, right? Well, using the assignment operator (=) certainly won't help:

```
var john = new Person(
  "John Smith",
  new Address("London Road", 123));

var jane = john;
jane.Name = "Jane Smith"; // John's name changed!
jane.Address.HouseNumber = 321; // John's address changed!
```

This does not work because now `john` and `jane` refer to the same object, so all changes to `jane` affect `john` too. What we want is `jane` to become a new, independent object, whose modifications do not affect `john` in any way.

ICloneable Is Bad

The .NET Framework comes with an interface called `ICloneable`. This interface has a single method, `Clone()`, but this method is *ill-specified*: the documentation does not suggest whether this should be a shallow copy or a deep copy. Also, the name of the method, `Clone`, does not really help here since we don't know exactly what cloning does. The typical implementation of `ICloneable` for a type (say, Person) is something like this:

```
public class Person : ICloneable
{
  // members as before
  public Person Clone()
  {
    return (Person)MemberwiseClone();
  }
}
```

The method `Object.MemberwiseClone()` is a protected method of `Object`, so it's automatically inherited by every single reference type. It creates a *shallow copy* of the object. In other words, if you were to implement it on `Address` and `Person` in our example, you'd hit the following problem:

```
var john = new Person(
  "John Smith",
  new Address("London Road", 123));

var jane = john.Clone();
jane.Name = "Jane Smith"; // John's name DID NOT change (good!)
jane.Address.HouseNumber = 321; // John's address changed :(
```

This helped, but not a lot. Even though the name was now assigned correctly, `john` and `jane` now share an `Address` reference – it was simply copied over, so they both point to the same `Address`. So shallow copy is not for us: we want *deep copying*, i.e., recursive copying of all object's members and the construction of shiny new counterpart objects, each initialized with identical data.

Deep Copying with a Special Interface

If you want to have an interface specifically to indicate that your objects support the notion of deep copying, I recommend you be explicit about it, i.e.,

```
interface IDeepCopyable<T>
{
  T DeepCopy();
}
```

where T is the type of object to clone. Here is an example implementation:

```
public class Person : IDeepCopyable<Person>
{
  public string[] Names;
  public Address Address;

  public Person DeepCopy()
  {
    var copy = new Person();
    copy.Names = Array.Copy(Names);    // string[] is not IDeepCopyable
    copy.Address = Address.DeepCopy(); // Address is IDeepCopyable
    return copy;
  }
  // other members here
}
```

You'll notice that, in the implementation of `DeepCopy()`, we adopt different strategies depending on whether or not the members are themselves `IDeepCopyable`. If they are, things are fairly straightforward. If they are not, we need to use an appropriate deep-copy mechanic for the given type. For example, for an array, you would call `Array.Copy()`.

This has two benefits compared to `ICloneable`:

- It is explicit in its intent: it talks specifically about deep copying.
- It is strongly typed, whereas `ICloneable` returns an `object` that you are expected to cast.

Deep Copying Objects

What we are going to disuss is how to perform deep copying of various fundamental .NET data types.

Value types such as `int`, `double`, and similar, as well as anything that's a `struct` (`DateTime`, `Guid`, `Decimal` etc.) can be deep-copied using copy assignment:

```
var dt = new DateTime(2016, 1, 1);
var dt2 = dt; // deep copy!
```

The `string` type is a bit special: even though it is a reference type, it is also immutable, meaning the value of a particular string cannot be changed: all we can do is reassign a *reference* that points to some string or other. The consequence is that, when deep-copying individual strings, we can continue to happily use the = operator:

```
string s = "hello";
string w = s; // w refers to "hello"
w = "world";  // w refers to "world"
Console.WriteLine(s); // still prints "hello"
```

And then there are data structures you do not control. For example, arrays can be copied using `Array.Copy()`. To deep-copy a `Dictionary<>`, you can use its copy constructor:

```
var d = new Dictionary<string, int>
{
  ["foo"] = 1,
  ["bar"] = 2
};
var d2 = new Dictionary<string, int>(d);
d2["foo"] = 55;
Console.WriteLine(d["foo"]); // prints 1
```

But even a structure such as a `Dictionary` has no idea how to deep-copy reference types it contains. So, if you try to use this approach to deep-copy a `Dictionary<string, Address>`, you're going to be out of luck:

```
var d = new Dictionary<string, Address>
{
  ["sherlock"] = new Address {HouseNumber = 221, StreetName = "Baker St"}
};
var d2 = new Dictionary<string, Address>(d);
d2["sherlock"].HouseNumber = 222;
Console.WriteLine(d["sherlock"].HouseNumber); // prints "222"
```

Instead, you have to make sure that deep copying is performed on each value of the dictionary, e.g.:

```
var d2 = d.ToDictionary(x => x.Key, x => x.Value.DeepCopy());
```

The same goes for other collections: `Array.Copy` is fine if you're storing strings or integers, but for composite objects, it just won't do. This is where LINQ's various collection-generating operations, such as `ToArray()`/`ToList()`/`ToDictionary()`, come in very useful. Also, don't forget that even though you cannot make BCL types such as `Dictionary<>` implement the interfaces you want, you can still give them appropriate `DeepCopy()` members as extension methods.

Duplication via Copy Construction

The simplest way of implementing proper duplication is to implement *copy constructors*. A copy constructor is an artifact straight from the C++ world – it is a constructor that takes another instance of the type we're in, and copies that type into the current object, for example:

```
public Address(Address other)
{
  StreetAddress = other.StreetAddress;
  City = other.City;
  Country = other.Country;
}
```

And, similarly

```
public Person(Person other)
{
  Name = other.Name;
  Address = new Address(other.Address); // uses a copy constructor here
}
```

This allows us to perform a deep copy of `john` to `jane`:

```
var john = new Person(
  "John Smith",
  new Address("London Road", 123));

var jane = new Person(john); // copy constructor!
jane.Name = "Jane Smith";
jane.Address.HouseNumber = 321; // john is still at 123
```

You'll notice that, although strings are reference types, we don't have to perform any action to deep-copy them. This is because strings are immutable, and you cannot actually modify a string, only construct a new string and rebind the references. One less thing to worry about, eh?

But beware. If, for example, we had an *array* of names (i.e., `string [] names`), we *would* have to explicitly copy the entire array using `Array.Copy` because, guess what, arrays are mutable. The same goes for any other data type that isn't a primitive type, `string` or a `struct`.

Now, the copy-constructor is pretty good in that it provides a unified copying interface, but it is of little help if the client is unable to discover it. At least, when the developer sees an `IDeepCopyable` interface with a `DeepCopy()` method, they know what they are getting; discoverability of a copy constructor is suspect. Another problem with this approach is that it's very intrusive: it requires that every single class in the composition chain implement a copy constructor, and will likely malfunction if any class doesn't do it correctly. As such, it's a very challenging approach to use on preexisting data structures, as you'd be violating OCP on a massive scale if you wanted to support this post-hoc.

Serialization

We need to thank the designers of C# for the fact that most objects in C#, whether they be primitive types or collections, are ‘trivially serializable’ – that, by default, you should are able take a class and save it to a file or to memory without adding extra code to the class (well, maybe an attribute or two, at most) or having to fiddle with reflection.

Why is this relevant to the problem at hand? Because if you can serialize something to a file or to memory, you can then deserialize it, preserving all the information, including all the dependent objects. Isn’t this convenient? For example, you could define an extension method (this is formally known as a Behavioral Mixin) for in-memory cloning using binary serialization:

```
public static T DeepCopy<T>(this T self)
{
  using (var stream = new MemoryStream())
  {
    BinaryFormatter formatter = new BinaryFormatter();
    formatter.Serialize(stream, self);
    stream.Seek(0, SeekOrigin.Begin);
    object copy = formatter.Deserialize(stream);
    return (T) copy;
  }
}
```

This code simply takes an object of *any* type `T`, performs binary serialization into memory, then deserializes from that memory, thereby gaining a deep copy of the original object.

This approach is fairly universal, and will let your easily clone your objects:

```
var foo = new Foo { Stuff = 42, Whatever = new Bar { Baz = "abc"} };
var foo2 = foo.DeepCopy();
foo2.Whatever.Baz = "xyz"; // works fine
```

There is just one catch: binary serialization requires every class to be marked with `[Serializable]`, otherwise the serializer simply throws an exception (not a good thing). So, if we wanted to use this approach on an existing set of classes, including those *not* marked as `[Serializable]`, we might go with a different approach that doesn't require the aforementioned attribute. For example, you could use XML serialization instead:

```
public static T DeepCopyXml<T>(this T self)
{
  using (var ms = new MemoryStream())
  {
    XmlSerializer s = new XmlSerializer(typeof(T));
    s.Serialize(ms, self);
    ms.Position = 0;
    return (T) s.Deserialize(ms);
  }
}
```

You can use any serializer you want, the only requirement is that it knows how to traverse every single element in the object graph. Most serializers are smart enough to go over things that shouldn't be serialized (like read-only properties), but sometimes they need a little help in order to make sense of trickier structures. For example, the XML serializer will not serialize an `IDictionary`, so if you are using a dictionary in your class, you'd need to mark it as `[XmlIgnore]` and create a Property Surrogate that we discuss in the Adapter chapter.

Prototype Factory

If you have predefined objects that you want to replicate, where do you actually store them. A static field of some class? Perhaps. In fact, suppose our company has both main and auxiliary offices. Now we could try to declare some static variables, e.g.:

```
static Person main = new Person(null,
  new Address("123 East Dr", "London", 0));
static Person aux = new Person(null,
  new Address("123B East Dr", "London", 0));
```

We could stick these members into `Person` so as to provide a hint that, when you need a person working at a main office, just clone `main`, and similarly for the auxiliary office, one can clone `aux`. But this is far fron intuitive: what if we want to prohibit construction of people working anywhere other than these two offices? And, from the SRP perspective, it would also make sense to keep the set of possible addresses separate.

This is where a Prototype Factory comes into play. Just like an ordinary factory, it can store these static members and provide convenience methods for creating new employees:

```
public class EmployeeFactory
{
  private static Person main =
    new Person(null, new Address("123 East Dr", "London", 0));
  private static Person aux =
    new Person(null, new Address("123B East Dr", "London", 0));

  public static Person NewMainOfficeEmployee(string name, int suite) =>
    NewEmployee(main, name, suite);

  public static Person NewAuxOfficeEmployee(string name, int suite) =>
    NewEmployee(aux, name, suite);

  private static Person NewEmployee(Person proto, string name, int suite)
  {
    var copy = proto.DeepCopy();
    copy.Name = name;
    copy.Address.Suite = suite;
    return copy;
  }
}
```

Notice how, following the DRY principle, we don't call `DeepCopy()` in more than one location: all the different `NewXxxEmployee()` methods simply forward their arguments to one private `NewEmployee()` method, passing it the prototype to use when constructing a new object.

The above prototype factory can now be used as:

```
var john = EmployeeFactory.NewMainOfficeEmployee("John Doe", 100);
var jane = EmployeeFactory.NewAuxOfficeEmployee("Jane Doe", 123);
```

Naturally, this implementation assumes that the constructors of `Person` are accessible; if you want to keep them `private/protected`, you'll need to implement the Inner Factory approach as outlined in the *Factories* chapter.

Summary

The Prototype design pattern embodies the notion of *deep* copying of objects so that, instead of doing full initialization each time, you can take a premade object, copy it, fiddle it a little bit and then use it independently of the original.

There are really only two ways of implementing the Prototype pattern. They are:

- Writing code that correctly duplicates your object, i.e., performs a deep copy. This can be done in a copy constructor, or you can define an appropriately named method, possibly with a corresponding interface (but *not* `ICloneable`).
- Write code for the support of serialization/deserialization and then use this mechanism to implement cloning as serialization immediately followed by deserialization. This carries the extra computational cost; its significance depends on how often you need to do the copying. The advantage of this approach is that you can get away without significantly modifying existing structures. It's also much safer, because you're less likely to forget to clone a member properly.

Don't forget, for value types, the cloning problem doesn't really exist: if you want to clone a `struct`, just assign it to a new variable. Also, strings are immutable, so you can use the assignment operator = on them without worrying that subsequent modification will affect more objects than it should.

Singleton

> When discussing which patterns to drop, we found that we still love them all. (Not really – I'm in favor of dropping Singleton. Its use is almost always a design smell.) – *Erich Gamma*

The Singleton is by far the most hated design pattern in the (rather limited) history of design patterns. Just stating that fact, however, doesn't mean you shouldn't use the singleton: a toilet brush is not the most pleasant device either, but sometimes it is simply necessary.

The Singleton design pattern grew out of a very simple idea that you should only have one instance of a particular component in your application. For example, a component that loads a database into memory and offers a read-only interface is a prime candidate for a Singleton since it really doesn't make sense to waste memory storing several identical datasets. In fact, your application might have constraints such that 2 or more instances of the database simply won't fit into memory, or will result in such a lack of memory as to cause the program to malfunction.

Singleton by Convention

The na☒ve approach to this problem is to simply agree that we are not going to instantiate this object more than once, i.e.:

```
public class Database
{
  /// <summary>
  /// Please do not create more than one instance.
  /// </summary>
  public Database() {}
};
```

The problem with this approach, apart from the fact that your developer colleagues might simply ignore the advice, is that objects can be created in stealthy ways where the call to the constructor isn't immediately obvious. This can be anything – a call through reflection, creation in a factory (e.g., `Activator.CreateInstance`), or injection of the type by an IoC container.

The most obvious idea that comes to mind is to offer a single, static global object:

```
public static class Globals
{
  public static Database Database = new Database();
}
```

However, this really doesn't do much in terms of safety: clients are not in any way prevented from constructing additional `Databases` as they see fit. And how will the client find the `Globals` class?

Classic Implementation

So now that we know what the problem is, how can we turn life sour for those interested in making more than one instance of an object? Simply put a static counter right in the constructor and `throw` if the value is ever incremented:

```
public class Database
{
  private static int instanceCount = 0;
  Database()
  {
    if (++instanceCount > 1)
      throw new InvalidOperationException("Cannot make >1 database!");
  }
};
```

This is a particularly hostile approach to the problem: even though it prevents the creation of more than one instance by throwing an exception, it fails to *communicate* the fact that we don't want anyone calling the constructor more than once. Even if you adorn it with plenty of XML documentation, I guarantee there will still be some poor soul trying to call this more than once in some non-determinstic setting. Probably in production, too!

The only way to prevent explicit construction of Database is to make its constructor private and introduce a property or method to return the one and only instance:

```
public class Database
{
  private Database() { ... }
  public static Database Instance { get; } = new Database();
}
```

Note how we removed the possibility of directly creating Database instances by hiding the constructor. Of course, you can use reflection to access private members, so construction of this class isn't quite impossible, but it does require extra hoops to jump through, and hopefully this is enough to prevent most people trying to construct one.

By declaring the instance as static, we removed any possibility of controlling the lifetime of the database: it now lives as long as the program does.

Lazy Loading and Thread Safety

The implementation shown in the previous section happens to be thread-safe. After all, static constructors are guaranteed to run only once per AppDomain, before any instances of the class are created or any static members accessed.

But what if you don't want initialization in the static constructor? What if, instead, you want to initialize the singleton (i.e., call its constructor) only when the object is first accessed? In this case, you can use Lazy<T>[12]:

[12]Note that, similar to C#, F#'s default implementation also uses lazy. The only difference is that F# has a somewhat more concise syntax: writing lazy(x + y()) automatically constructs a Lazy<'T> behind the scenes.

```
public class MyDatabase
{
  private MyDatabase()
  {
    Console.WriteLine("Initializing database");
  }
  private static Lazy<MyDatabase> instance =
    new Lazy<MyDatabase>(() => new MyDatabase());

  public static MyDatabase Instance => instance.Value;
}
```

This is also a thread-safe approach because the objects `Lazy<T>` creates are thread-safe by default. In a multi-threaded setting, the first thread to access the `Value` property of a `Lazy<T>` is the one that initializes it for all subsequent accesses on all threads.

The Trouble with Singleton

Let us now consider a concrete example of a Singleton. Suppose that our database contains a list of capital cities and their populations. The interface that our singleton database is going to conform to is:

```
public interface IDatabase
{
  int GetPopulation(string name);
}
```

We have a single method that gives us the population of a given city. Now, let us suppose that this interface is adopted by a concrete implementation called `SingletonDatabase` that implements the Singleton the same way as we've done before:

```
public class SingletonDatabase : IDatabase
{
  private Dictionary<string, int> capitals;
  private static int instanceCount;
  public static int Count => instanceCount;

  private SingletonDatabase()
  {
    WriteLine("Initializing database");

    capitals = File.ReadAllLines(
      Path.Combine(
        new FileInfo(typeof(IDatabase).Assembly.Location).DirectoryName,
          "capitals.txt")
      )
      .Batch(2) // from MoreLINQ
      .ToDictionary(
        list => list.ElementAt(0).Trim(),
        list => int.Parse(list.ElementAt(1)));
  }

  public int GetPopulation(string name)
  {
    return capitals[name];
  }

  private static Lazy<SingletonDatabase> instance =
    new Lazy<SingletonDatabase>(() =>
    {
      instanceCount++;
      return new SingletonDatabase();
    });

  public static IDatabase Instance => instance.Value;
}
```

The constructor of the database reads the names and populations of various capitals from a text file and stores them in a `Dictionary<>`. The `GetPopulation()` method is used as an accessor to get the population of a given city.

As we noted before, the real problem with singletons like the one above is their use in other components. Here's what I mean: suppose that, on the basis of the above, we build a component for calculating the sum total population of several different cities:

```
public class SingletonRecordFinder
{
  public int TotalPopulation(IEnumerable<string> names)
  {
    int result = 0;
    foreach (var name in names)
      result += SingletonDatabase.Instance.GetPopulation(name);
    return result;
  }
}
```

The trouble is that `SingletonRecordFinder` is now firmly dependent on `SingletonDatabase`. This presents an issue for testing: if we want to check that `SingletonRecordFinder` works correctly, we need to use data from the actual database, i.e.:

```
[Test]
public void SingletonTotalPopulationTest()
{
  // testing on a live database
  var rf = new SingletonRecordFinder();
  var names = new[] {"Seoul", "Mexico City"};
  int tp = rf.TotalPopulation(names);
  Assert.That(tp, Is.EqualTo(17500000 + 17400000));
}
```

This is a terrible unit test. It tries to read a live database (something that you typically don't want to do too often), but it's also very fragile, because it depends on the concrete values in the database. What if the population of Seoul changes (as a result of North Korea opening its borders, perhaps)? Then the test will break. But of course, many people run tests on Continuous Integration systems that are isolated from live databases, so that fact makes the approach even more dubious.

This test is also bad for ideological reasons. Remember, we want a *unit* test where the unit we're testing is the `SingletonRecordFinder`. However, the test above is not a unit test but an *integration* test because the record finder uses `SingletonDatabase`, so in effect we're testing both systems at the same time. Nothing wrong with that if an integration test is what you wanted, but we would really prefer to test the record finder in isolation.

So we know we don't want to use an actual database in a test. Can we replace the database with some dummy component that we can control from within our tests? Well, in our current design, this is impossible, and it is precisely this inflexibility that is the Singeton's downfall.

So, what can we do? Well, for one, we need to stop depending on `Singleton-Database` explicitly. Since all we need is something implementing the `Database` interface, we can create a new `ConfigurableRecordFinder` that lets us configure where the data comes from:

```
public class ConfigurableRecordFinder
{
  private IDatabase database;

  public ConfigurableRecordFinder(IDatabase database)
  {
    this.database = database;
  }

  public int GetTotalPopulation(IEnumerable<string> names)
  {
    int result = 0;
    foreach (var name in names)
      result += database.GetPopulation(name);
    return result;
  }
}
```

We now use the `database` reference instead of using the singleton explicitly. This lets us make a dummy database specifically for testing the record finder:

```
public class DummyDatabase : IDatabase
{
  public int GetPopulation(string name)
  {
    return new Dictionary<string, int>
    {
      ["alpha"] = 1,
      ["beta"] = 2,
      ["gamma"] = 3
    }[name];
  }
}
```

And now, we can rewrite our unit test to take advantage of this `DummyDatabase`:

```
[Test]
public void DependentTotalPopulationTest()
{
  var db = new DummyDatabase();
  var rf = new ConfigurableRecordFinder(db);
  Assert.That(
    rf.GetTotalPopulation(new[]{"alpha", "gamma"}),
    Is.EqualTo(4));
}
```

This test is more robust because if data changes in the actual database, we won't have to adjust our unit test values – the dummy data stays the same. Also, it opens interesting possibilities. We can now run tests against an empty database or, say, a database whose size is greater than the available RAM. You get the idea.

Singletons and Inversion of Control

The approach with explicitly making a component a singleton is distinctly invasive, and a decision to stop treating the class as a Singleton down the line will end up particularly costly. An alternative solution is to adopt a convention where, instead of directly enforcing the lifetime of a class, this function is outsourced to an Inversion of Control (IoC) container.

Here's what defining a singleton component looks like when using the Autofac dependency injection framework:

```
var builder = new ContainerBuilder();
builder.RegisterType<Database>().SingleInstance(); // <-- singleton!
builder.RegisterType<RecordFinder>();

var container = builder.Build();
var finder = container.Resolve<RecordFinder>();
var finder2 = container.Resolve<RecordFinder>();
WriteLine(ReferenceEquals(finder, finder2)); // True
// finder and finder2 refer to the same database
```

Many people believe that using a singleton in a DI container is the only socially acceptable use of a singleton. At least, with this approach, if you need to replace a singleton object with something else, you can do it in one central place: the container configuration code. An added benefit is that you won't have to implement any singleton logic yourself, which prevents possible errors. Oh and did I mention that all container operations in Autofac are thread-safe?

In actual fact, one thing IoC containers highlight is the fact that a Singleton is only a unique case of lifetime management (one object per lifetime of entire application). Different lifetimes are possible – you can have one object per thread, one object per web request, and so on. You can also have *pooling* – situations where the number of live object instances can be between 0 and X, whatever X happens to be.

Monostate

Monostate is a variation on the Singleton pattern. It is a class that *behaves* like a singleton while appearing as an ordinary class.

For example, suppose you are modeling a company structure, and a company typically has only one CEO. What you can do is define the following class:

```
public class ChiefExecutiveOfficer
{
  private static string name;
  private static int age;

  public string Name
  {
    get => name;
    set => name = value;
  }

  public int Age
  {
    get => age;
    set => age = value;
  }
}
```

Can you see what's happening here? The class appears as an ordinary class with getters and setters, but they actually work on `static` data!

This might seem like a really neat trick: you let people instantiate `ChiefExecutiveOfficer` as many times as they want, but all the instances refer to the same data. However, how are users supposed to know this? A user will happily instantiate two CEOs, assign them different `id`s and will be very surprised when both of them are identical!

The Monostate approach works to some degree and has a couple of advantages. For example, it is easy to inherit, it can leverage polymorphism, and its lifetime is reasonably well-defined (but then again, you might not always wish it so). Its greatest advantage is that you can take an existing object that's already used throughout the system, patch it up to behave in a Monostate way, and provided your system works fine with the non-plurality of object intances, you've got yourself a Singleton-like implementation with no extra code needing to be rewritten.

But that's all that Monostate really is: a band-aid when you want one component to become a singleton throughout the entire codebase without any large-scale changes. This pattern is not meant for production, as it can cause too much confusion. If you need centralized control over things, a DI container is your best bet.

Multiton

A Multiton, like its name suggests, is a pattern that, instead of forcing us to have just one instances, gets us to have a finite number of named instances of some particular component. For example, suppose we have two subsystems – the main one, and another for backup:

```
enum Subsystem
{
  Main,
  Backup
}
```

If only a single printer is meant to exist for every subsystem, we can define the `Printer` class as follows:

```
class Printer
{
  private Printer() { }

  public static Printer Get(Subsystem ss)
  {
    if (instances.ContainsKey(ss))
      return instances[ss];

    var instance = new Printer();
    instances[ss] = instance;
    return instance;
  }

  private static readonly Dictionary<Subsystem, Printer> instances
    = new Dictionary<Subsystem, Printer>();
}
```

As before, we've hidden the constructor and made an accessor method that lazily constructs and returns a printer corresponding to the required subsystem. The above implementation is, of course, not thread-safe, but that can be easily corrected via the use of, say, a `ConcurrentDictionary`.

Notice also that the above implementation has the same problems as the Singleton in terms of direct dependencies. If your code relies on `Printer.Get(Subsystem.Main)`, how could you sustitue the result with a different implementation? Well, just as with the database example we looked at, the best solution would be to extract some `IPrinter` interface and to depend on that instead.

Summary

Singletons aren't totally evil but, when used carelessly, they'll mess up the testability and refactorability of your application. If you really must use a singleton, try avoiding using it directly (as in, writing `SomeComponent.Instance.Foo()`) and instead keep specifying it as a dependency (e.g., a constructor argument) where all dependencies are satisfied from a single location in your application (e.g., an Inversion of Control container). Relying on abstractions (interfaces/abstract classes) conforms with DIP and is generally a good idea if you want to perform a substitution later on.

Structural Patterns

As the name suggests, Structural patterns are all about setting up the structure of your application so as to improve SOLID conformance as well as general usability and maintainability of your code.

When it comes to determining the structure of an object, we can apply two fairly well-known methods:

- Inheritance: an object automagically acquires all members base class or classes. To allow instantiation, the object must implement every abstract member from its parent; if it does not, it is abstract and cannot be created (but you can inherit from it).
- Composition: generally implies that the child cannot exist without the parent. This is typically implemented with nested classes. For example, a class `Car` can have a nested class `Wheel`.
- Aggregation: an object can contain another object, but that object can also exist independently. Think of a `Car` having a `Person driver` field or property.

Nowadays, both composition and aggregation are treated in an identical fashion. If you have a `Person` class with a field of type `Address`, you have a choice as to whether `Address` is an external type or a nested type. In either case, provided it's `public`, you can instantiate it as either `Address` or `Person.Address`.

I would argue that using the word *composition* when we really mean aggregation has become so commonplace that we may as well use them in interchangeable fashion. Here's some proof: when we talk about IoC containers, we speak of a *composition root*. But wait, doesn't the IoC container control the lifetime of each object individually? It does, and so we're using the word 'composition' when we really mean 'aggregation'.

There are, fundamentally, three ways in which data structures can be defined in C#:

- *Statically*, when you simply write the classes and they get compiled. This is the most common case out there.
- *Via code generation*. This happens when structures get created from T4 templates or databases or some user scripts. Plenty of code gets generated behind the scenes when you edit UI, e.g., in a WinForms or WPF application.
- *Dynamically*, i.e., at runtime. This is the most sophisticated option. Advanced libraries are capable of constructing data structures and compiling them into executable code right at the moment when the application is executing. This approach is sometimes leveraged in design patterns and gives rise to their static-dynamic duality.

It's worth noting that many data structures are created implicitly behind the scenes by the compiler. This includes things like `ValueTuples`, classes for anonymous types, as well as state machines in charge of orchestrating enumerators (`yield` functionality) or asynchronous (`async/await`) operations.

In F#, the amount of data structures (and corresponding allocations) created behind the scenes is *huge*, as is the complexity of those data structures. For example, any sort of currying operations generate deep inheritance hierarchies. Or, let's say, you decide to pass an operator such as `(+)` as a parameter to a function – in this case, an entire `struct` will be created purely for the purpose of housing a method that would in turn call this operator. And it would be vary naive to think that all such allocations are automatically inlined during JIT compilation!

Adapter

I used to travel quite a lot, and quite often only when you arrive in a new country you remember that their sockets are different and you didn't prepare for this. This is why airport travel shops carry travel adapters, and also why some hotes (the better ones) have at least one outlet of a non-local type just in case a customer has forgotten to get an adapter, but needs to, say, work on their laptop without interruption.

A travel adapter that lets me plug a European plug into a UK or USA socket [13] is very good analogy to what's going on with the Adapter pattern in the software world: we are given an interface, but we want a different one, and building an adapter over the interface is what gets us to where we want to be.

Scenario

Suppose you're working with a library that's great at drawing pixels. You, on the other hand, work with geometric objects - lines, rectangles, that sort of thing. You want to keep working with those objects but also need the rendering, so you need to *adapt* your vector geometry to pixel-based representation.

Let us begin by defining the (rather simple) domain objects of our example:

[13] Just in case you're European like me and want to complain that everyone should be using European plugs and sockets: *no*, the UK plug design is technically better and safer, so if we did want just one standard, the UK one would be the one to go for.

```
public class Point
{
  public int X, Y;
  // other members omitted
}

public class Line
{
  public Point Start, End;
  // other members omitted
}
```

Let's now theorize about vector geometry. A typical vector object is likely to be defined by a collection of `Line` objects. Thus, we can make a class that would simply inherit from `Collection<Line>`:

```
public abstract class VectorObject : Collection<Line> {}
```

So, this way, if you want to define, say, a `Rectangle`, you can simply inherit from this type and there's no need to define additional storage:

```
public class VectorRectangle : VectorObject
{
  public VectorRectangle(int x, int y, int width, int height)
  {
    Add(new Line(new Point(x,y), new Point(x+width, y) ));
    Add(new Line(new Point(x+width,y), new Point(x+width, y+height) ));
    Add(new Line(new Point(x,y), new Point(x, y+height) ));
    Add(new Line(new Point(x,y+height), new Point(x+width, y+height) ));
  }
}
```

Now, here's the set-up. Suppose we want to draw lines on screen. Rectangles, even! Unfortunately, we cannot, because the only interface for drawing is literally this:

```
// the interface we have
public static void DrawPoint(Point p)
{
  bitmap.SetPixel(p.X, p.Y, Color.Black);
}
```

I'm using the `Bitmap` class here for illustration, but the actual implementation doesn't matter. Let's just take this at face value: we only have an API for drawing pixels. That's it.

Adapter

All right, so let's suppose we want to draw a couple of rectangles:

```
private static readonly List<VectorObject> vectorObjects
  = new List<VectorObject>
{
  new VectorRectangle(1, 1, 10, 10),
  new VectorRectangle(3, 3, 6, 6)
};
```

In order to draw these objects, we need to convert every one of them from a series of lines into a rather large number of points, because the only interface that we have for drawing is a `DrawPoint()` method. For this, we make a separate class that will store the points and expose them as a collection. That's right – this is our Adapter pattern!

```
public class LineToPointAdapter : Collection<Point>
{
  private static int count = 0;

  public LineToPointAdapter(Line line)
  {
    WriteLine($"{++count}: Generating points for line"
        + $" [{line.Start.X},{line.Start.Y}]-"
        + $"[{line.End.X},{line.End.Y}] (no caching)");
    int left = Math.Min(line.Start.X, line.End.X);
    int right = Math.Max(line.Start.X, line.End.X);
    int top = Math.Min(line.Start.Y, line.End.Y);
    int bottom = Math.Max(line.Start.Y, line.End.Y);

    if (right - left == 0)
    {
      for (int y = top; y <= bottom; ++y)
      {
        Add(new Point(left, y));
      }
    } else if (line.End.Y - line.Start.Y == 0)
    {
      for (int x = left; x <= right; ++x)
      {
        Add(new Point(x, top));
      }
    }
  }
}
```

The code above is simplified: we only handle perfectly vertical or horizontal lines and ignore everything else. The conversion from a line to a number of points happens right in the constructor, so our adapter is *eager*; don't worry, we'll make it lazy towards the end of this chapter.

We can now use this adapter to actually render some objects. We take the two rectangles from earlier and simply render them like this:

```
private static void DrawPoints()
{
  foreach (var vo in vectorObjects)
  {
    foreach (var line in vo)
    {
      var adapter = new LineToPointAdapter(line);
      adapter.ForEach(DrawPoint);
    }
  }
}
```

Beautiful! All we do is, for every vector object, get each of its lines, construct a `LineToPointAdapter` for that line and then iterate the set of points produced by the adapter, feeding them to `DrawPoint()`. And it works! (Trust me, it does.)

Adapter Temporaries

There's a major problem with our code, though: `DrawPoints()` gets called on literally every screen refresh that we might need, which means the same data for same line objects gets regenerated by the adapter, like, a zillion times. What can we do about it?

Well, on one hand, we can make some lazy-loading method, for example:

```
private static List<Point> points = new List<Point>();
private static bool prepared = false;

private static void Prepare()
{
  if (prepared) return;
  foreach (var vo in vectorObjects)
  {
    foreach (var line in vo)
    {
      var adapter = new LineToPointAdapter(line);
      adapter.ForEach(p => points.Add(p));
    }
  }
  prepared = true;
}
```

and then the implementation of `DrawPoints()` simplifies to

```
private static void DrawPointsLazy()
{
  Prepare();
  points.ForEach(DrawPoint);
}
```

But let's suppose, for a moment, that the original set of `vectorObjects` can change. Saving those points forever makes no sense then, but we still want to avoid the incessant regeneration of potentially repeating data. How do we deal with this? With caching, of course!

First of all, to avoid regeneration, we need unique ways of identifying lines, which transitively means we need unique ways of identifying points. ReSharper's **Generate | Equality Members** to the rescue:

```
public class Point
{
  // other members here

  protected bool Equals(Point other) { ... }
  public override bool Equals(object obj) { ... }

  public override int GetHashCode()
  {
    unchecked { return (X * 397) ^ Y; }
  }
}

public class Line
{
  // other members here
  protected bool Equals(Line other) { ... }
  public override bool Equals(object obj) { ... }

  public override int GetHashCode()
  {
    unchecked
    {
      return ((Start != null ? Start.GetHashCode() : 0) * 397)
        ^ (End != null ? End.GetHashCode() : 0);
    }
  }
}
```

As you can see, ReSharper (or Rider, if you prefer that IDE) has generated different implementations of `Equals()` as well as `GetHashCode()`. The latter is more important because it allows us to uniquely (to some degree) identify an object by its hash code without peforming a direct comparison. Now, we can build a new `LineToPointCachingAdapter` such that it caches the points and regenerates them only when necessary, i.e., when their hashes differ. The implementation is almost the same except for the following nuances.

First, the adapter now has a `static` cache of points that correspond to particular lines:

```
static Dictionary<int, List<Point>> cache
  = new Dictionary<int, List<Point>>();
```

The type `int` here is precisely the type returned from `GetHashCode()`. Now, when processing a `Line` in the constructor, we first check whether or not the line is already cached: if it is, we don't need to do anything:

```
hash = line.GetHashCode();
if (cache.ContainsKey(hash)) return; // we already have it
```

Notice that we actually *store* the hash of the current adapter in its non-static field. This allows us to store and use adapters that correspond to individual lines. Alternatively, we could make the entire adapter `static`.

The full implementation of the constructor is as before, except that instead of calling Add() for the generated points, we simply add them to the cache:

```
public LineToPointAdapter(Line line)
{
  hash = line.GetHashCode();
  if (cache.ContainsKey(hash)) return; // we already have it

  List<Point> points = new List<Point>();

  // points are added to the 'points' member as before, then...

  cache.Add(hash, points);
}
```

Lastly, we need to implement IEnumerable<Point>. This is easy: we use the hash field to reach into the cache and yield the right set of points:

```
public IEnumerator<Point> GetEnumerator()
{
  return cache[hash].GetEnumerator();
}
```

Yay! Thanks to hash functions and caching, we've drastically cut down on the number of conversions being made. The only issue with this implementation is that, in a long-running program, the cache can accumulate a huge number of unnecessary point collections. How would you clean it up? One idea would be to set up a timer to wipe the entire cache at regular intervals. See if you can come up with other possible solutions to this problem.

The Problem with Hashing

One reason why we implemented the adapter the way we did is that our current implementation is robust with respect to changes in objects. If any aspect of a Rectangle changes, the adapter will calculate a different hash value and will regenerate the appropriate set of points.

This is effectively done by *polling*: any time an adapted dataset is required, we take the target object and recalculate its hash. The assumption is always that the hash can be calculated quickly, and that hash collisions – situations where two different objects have identical hashes – are unlikely. Let's remind ourselves of how Point hashes are calculated:

```
public override int GetHashCode()
{
  unchecked
  {
    return (X * 397) ^ Y;
  }
}
```

The truth is, the hash function for a Point is a pretty bad hash function and will give us lots of collisions. For example, points (0,0) and (1, 397) will give the same hash value of 0 which means, in turn, that two lines with these Start points and an identical End point will end up overwriting each other's generated set of points with incorrect data, inevitably causing problems.

How would you solve this issue? Well, you could pick a prime number N that is larger than 397. That way, if you can guarantee that your values are less than this larger N, you won't have any collisions. Alternatively, you could go for a more robust hashing function. In the case of a Point, assuming positive X and Y, this could be as simple as

```
public long MyHashFunction()
{
  return (X << 32) | Y;
}
```

If you really wanted to preserve the GetHashCode() interface (it returns an int, remember), you could do it by downgrading the co-ordinates to a short – its range is plenty enough for a screen co-ordinate (until we get 64K screens, that is). Finally, there are plenty of sophisticated functions out there (Cantor pairing function, Szudzik function, etc.) that are able to handle situations at the boundaries of the range of numbers.

The point I'm trying to make here is that the calculation of hashing functions is a slippery slope: the code generated by IDEs might not be as robust as you think. What can we do to avoid all of this? Why, we can hold a reference to the adaptee, rather than the hash, in our cache. It's as simple as:

```
public class LineToPointAdapter
  : IEnumerable<Point>
{
  static Dictionary<Line, List<Point>> cache
    = new Dictionary<Line, List<Point>>();
  private Line line;

  public LineToPointAdapter(Line line)
  {
    if (cache.ContainsKey(line)) return; // we already have it
    this.line = line;

    // as before

    cache.Add(line, points);
  }

  public IEnumerator<Point> GetEnumerator()
  {
    return cache[line].GetEnumerator();
  }
}
```

What's the difference? Well, the difference is that, when searching through the dictionary, *both* `GetHashCode()` and `Equals()` are used to find the right item. As a result, collisions can still occur, but they won't mess up the final value. This approach does have its downsides though: for example, the lifetime of lines is now bound to the adapter because it has strong references to them.

This approach of holding on to the reference gives us an additional benefit: laziness. Instead of calculating everything in the constructor, we can split the preparation of points into a separate function that only gets invoked when adapter points are iterated:

```
public class LineToPointAdapter : IEnumerable<Point>
{
  ...
  private void Prepare()
  {
    if (cache.ContainsKey(line)) return; // we already have it
    // rest of code as before
  }

  public IEnumerator<Point> GetEnumerator()
  {
    Prepare();
    return cache[line].GetEnumerator();
  }
}
```

Property Adapter (Surrogate)

One very common application of the Adapter design pattern is to get your class to provide additional properties that serve only one purpose: to take existing fields or properties and expose them in some useful way, quite often as projections to a different data type. And while it often makes sense to do this in a separate class (for example, when building a ViewModel), sometimes you're forced to do this right in the class where the original data is kept.

Consider the following example: if you have an `IDictionary` member in your class, you cannot use an `XmlSerializer` because Microsoft didn't implement this functionality "due to schedule constraints". Consequently, if you want a serializable dictionary, you have two options: you either go online and search for a `SerializableDictionary` implementation or, alternatively, you build a property adapter (or surrogate) which exposes the dictionary in a way that's easy to serialize.

For example, suppose you need to serialize the following property:

```
public Dictionary<string, string> Capitals { get; set; }
```

To make it happen, you would first of all mark the property as `[XmlIgnore]`. You would then construct another property of a type that *can* be serialized, such as an array of tuples:

```
public (string, string)[] CapitalsSerializable
{
  get
  {
    return Capitals.Keys.Select(country =>
      (country, Capitals[country])).ToArray();
  }
  set
  {
    Capitals = value.ToDictionary(x => x.Item1, x => x.Item2);
  }
}
```

I have made very careful choice of types for serialization here:

- The overall type of the variable is an array. If you made this a `List`, the serializer would never call the setter, and would instead try to use the getter and then `Add()` to that getter – we definitely don't want that.
- We are using a `ValueTuple` instead of an ordinary `Tuple`. Conventional tuples cannot be serialized because they do not have parameterless constructors, whereas `ValueTuples` don't have this problem.

In case you're wondering, here's how a serialized class would look in XML:

```
<?xml version="1.0" encoding="utf-16"?>
<CountryStats xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <CapitalsSerializable>
    <ValueTupleOfStringString>
      <Item1>France</Item1>
      <Item2>Paris</Item2>
    </ValueTupleOfStringString>
  </CapitalsSerializable>
</CountryStats>
```

The adapter presented here is quite different to what you might expect because the API we're trying to adapt the class to is *implicit* – serialization mechanics are concealed by the serializer we're using, so the only way to know about this issue is through trial and error[14].

This example is a bit ambiguous with respect to SOLID principles: on the one hand, we're separating out the serialization concern. On the other hand, should this really be part of the class itself? It would be a lot neater if we could decorate the member with some `[SerializeThisDictionary]` attribute and have the conversion process handled elsewhere. Alas, such are the limitations of the way serialization is implemented in .NET.

Generic Value Adapter

In C++, unlike in C#, generic arguments don't have to be types: they can be literals instead. For example, you can write `template <int n> class X {}` and then instantiate a class of type `X<42>`. There are cases when this sort of functionality is necessary in C# and, even though the language does not allow the use of values in generic arguments, we can build adapters that help us adapt values to generic types.

The idea behind this is very simple so, to make it interesting, I'm going to throw in an added bonus: not only are we going to work with the Generic Value Adapter pattern, but we'll also make use of some advanced generic magic just to spice things up.

First, here's the scenario I propose. Say you're working in a mathematical or graphics domain and you want to have (geometric) vectors of different sizes and using different data types. For example, you want `Vector2i` to be a vector with two integer values, whereas a `Vector3f` would be a 3-dimensional vector with floating-point values.

What we really want is to have a class `Vector<T, D>` (`T` = type, `D` = dimensions) that would be defined as

[14]You really want to use third-party serialization components if you can. The support for both binary and XML serialization in .NET is very patchy and has lots of unpleasant caveats. If you were to take a JSON serializer such as JSON.NET, none of the problems described above would be an issue.

```
public class Vector<T, D>
{
  protected T[] data;

  public Vector()
  {
    data = new T[D]; // impossible
  }
}
```

and then instantiate it as

```
var v = new Vector<int, 2>(); // impossible
```

Both the constructor initialization and the instantiation are impossible in C#. The solution to this problem is not pretty: we basically wrap literals like 2, 3 and so on inside classes. To do this, first of all, we define an interface for returning an integer:

```
public interface IInteger
{
  int Value { get; }
}
```

And now this interface can be implemented by concrete classes that would yield values of 2, 3 and so on. To make it a bit neater to use, I'll put all of those inside a class that would act as an enum-like entity:

```
public static class Dimensions
{
  public class Two : IInteger
  {
    public int Value => 2;
  }

  public class Three : IInteger
  {
    public int Value => 3;
  }
}
```

So now we can finally define a working `Vector<T, D>` class that would intialize the data correctly:

```
public class Vector<T, D>
  where D : IInteger, new()
{
  protected T[] data;

  public Vector()
  {
    data = new T[new D().Value];
  }
}
```

This approach may be convoluted but it does, in fact, work. We require that `D` is an `IInteger` that also has a default constructor and, when it comes to initializing the data storage, we spin up an instance of `D` and take its value.

To make use of this new class, you'd write something like the following:

```
var v = new Vector<int, Dimensions.Two>();
```

Alternatively, you could make things reusable by defining inheriting types, for example:

```
public class Vector2i : Vector<int, Dimensions.Two> {}
// and then
var v = new Vector2i();
```

So that's it, we are done with the discussion of the Generic Value Adapter pattern as such. As I'm sure you can appreciate, the idea is both trivial and, unfortunately, ugly at the same time. Imagine having to make `Dimensions.Three`, `Dimensions.Four` and so on! Maybe some code generation can help here, though.

Now, it would be grossly unfair of me to abandon this example and just let you fend for yourself from this point on, so let's discuss a few ideas on how to get this example to production status. Even though these ideas are not central to design patterns, an attempt to get this `Vector` fully functional brings up one of the tricky aspects of C# â€" namely, recursive generics.

Let's start with the obvious things: we want to somehow access and modify data in the vector as if it were an array. A simple approach would be to simply expose an indexer:

```
public class Vector<T, D>
  where D : IInteger, new()
{
  // ... other members omitted
  public T this[int index]
  {
    get => data[index];
    set => data[index] = value;
  }
}
```

Similarly, if you decide to inherit, you can create additional getters and setters for having named co-ordinates:

```
public class Vector2i : Vector<int, Dimensions.Two>
{
  public int X
  {
    get => data[0];
    set => data[0] = value;
  }
  // similarly for Y
}
```

You could, theoretically, stick predictable properties such as `X`, `Y`, `Z` into the base class, too, but that would be a bit confusing because then you could have a 1-dimensional vector with an exposed Z co-ordinate that would simply throw an exception when accessed.

Anyways, with this all set you can now initialize a vector as follows:

```
var v = new Vector2i();
v[0] = 123; // using an indexer
v.Y = 456;  // using a property
```

Of course, it would really be nice if we could somehow initialize data in the constructor. Thanks to the `params` keyword, our base `Vector` can have a constructor taking an arbitrary number of arguments. We just need to make sure that the data being initialized is of the right size:

```
public Vector(params T[] values)
{
  var requiredSize = new D().Value;
  data = new T[requiredSize];

  var providedSize = values.Length;

  for (int i = 0; i < Math.Min(requiredSize, providedSize); ++i)
    data[i] = values[i];
}
```

Now we can initialize a vector using a constructor; we cannot really initialize the derived `Vector2i` this way, though, not until we create a forwarding constructor:

```
public class Vector2i : Vector<int, Dimensions.Two>
{
  public Vector2i() {}
  public Vector2i(params int[] values) : base(values) {}
}
```

So now we can finally make a `new Vector2i(2, 3)` and everything will compile. This is actually one of the two possible approaches to instantiating these vectors, the other involving the use of a Factory Method. But, before we get there, let's first of all consider one problem that will throw a fairly big spanner in the works.

Here's what I want to be able to write:

```
var v = new Vector2i(1, 2);
var vv = new Vector2i(3, 2);
var result = v + vv;
```

Now, this is a sad story. We cannot go into our `Vector<T, D>` and give it an `operator +`. Why not? Well because `T` is not constrained to numerics. It could be a `Guid` or something, and the operation to add two GUIDs is undefined. There is no way for us to tell C# to constrain `T` to numeric types (other languages, such as Rust, have solved this problem), so the only way we can get this all to work is to create more types derived from `Vector` – types such as `VectorOfInt`, `VectorOfFloat`, and so on.

```
public class VectorOfInt<D> : Vector<int, D>
  where D : IInteger, new()
{
  public VectorOfInt() {}

  public VectorOfInt(params int[] values) : base(values) {}

  public static VectorOfInt<D> operator +
    (VectorOfInt<D> lhs, VectorOfInt<D> rhs)
  {
    var result = new VectorOfInt<D>();
    var dim = new D().Value;
    for (int i = 0; i < dim; i++)
    {
      result[i] = lhs[i] + rhs[i];
    }

    return result;
  }
}
```

As you can see from this listing, we've had to replicate the constructor API of `Vector`, but we managed to provide a good `operator +` implementation that adds the two vectors together. Now all we need to do is modify our Vector2i and we're good to go:

```
public class Vector2i : VectorOfInt<Dimensions.Two>
{
  public Vector2i(params int[] values) : base(values)
  {
  }
}
```

Notice something interesting: we removed the parameterless constructor because it's no longer required, as it's now contained in VectorOfInt. However, we still have to keep the params constructor so we can initialize a Vector2i instance.

Here's a final complication that we can consider. Suppose you're not really interested in doing this constructor propagation all over the place. Say you decide that, instead of constructors, all derived classes (VectorOfInt, VectorOfFloat, Vector2i etc.) will have *no* constructors in their bodies. Instead, we decide that the creation of all these types will be handled by a single Vector<T, D>.Create() factory method. How can we get this done?

This situation is not simple, and calls for the use of recursive generics. Why? Because the static Vector.Create() method needs to return the right type. If I call Vector3f.Create() I expect a Vector3f to be returned, not a Vector<float, Dimensions.Three> and not VectorOfFloat<Dimensions.Three> either.

This means that we need to make several modifications. First, Vector now gets a new generic parameter TSelf referring to the class deriving from it:

```
public abstract class Vector<TSelf, T, D>
  where D : IInteger, new()
  where TSelf : Vector<TSelf, T, D>, new()
{
  // ...
}
```

As you can see, TSelf is constrained to be an inheritor of Vector<TSelf, T, D>. Now, any derived type (say, VectorOfFloat) needs to be changed to

```
public class VectorOfFloat<TSelf, D>
  : Vector<TSelf, float, D>
  where D : IInteger, new()
  where TSelf : Vector<TSelf, float, D>, new()
{
  // wow, such empty!
}
```

Notice that this class no longer has any forwarding constructors, since we plan on using a factory method. Similarly, you'd have to modify any class that derives from `VectorOfFloat`, for example:

```
public class Vector3f
  : VectorOfFloat<Vector3f, Dimensions.Three>
{
  // empty again
}
```

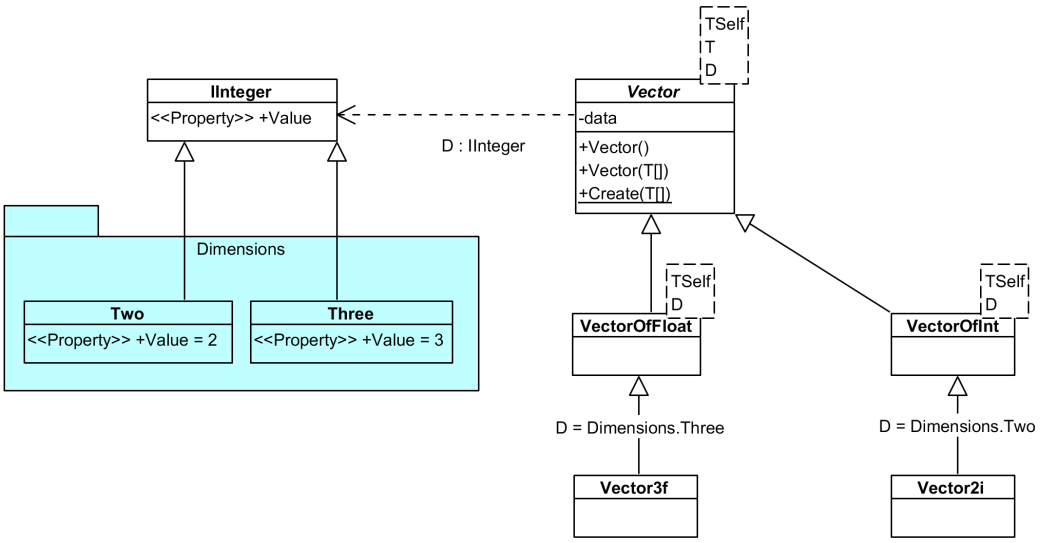
Notice how the `TSelf` gets propagated up the hierarchy: first, `Vector3f` travels up to `VectorOfFloat` and then up to `Vector`. This way, we can be sure that `Vector` knows that its factory method needs to return a `Vector3f`. Oh, and speaking of the factory method, we can finally write it!

```
public static TSelf Create(params T[] values)
{
  var result = new TSelf();
  var requiredSize = new D().Value;
  result.data = new T[requiredSize];

  var providedSize = values.Length;

  for (int i = 0; i < Math.Min(requiredSize, providedSize); ++i)
    result.data[i] = values[i];

  return result;
}
```

This is where `TSelf` comes in handy – it is the return type of our factory method. Now, whichever derived class you create, making an instance of this class is as simple as writing

```
var coord = Vector3f.Create(3.5f, 2.2f, 1);
```

And there you have it! Naturally, the type of `coord` above is a `Vector3f` – no need for casts or any other magic. This is the kind of functionality recursive generics allow you to have. Here is an illustration of our entire scenario:



Adapter in Dependency Injection

There are certain advanced Adapter scenarios that are handled nicely by Dependency Injection frameworks such as Autofac. The approach here is, admittedly, somewhat different to the “adapt component X to interface Y” that is discussed in the rest of the chapter.

Consider a scenario where your application has a bunch of commands that you want to invoke. Each command is able to execute itself, and that’s about it.

```
public interface ICommand
{
  void Execute();
}

public class SaveCommand : ICommand
{
  public void Execute()
  {
    Console.WriteLine("Saving current file");
  }
}

public class OpenCommand : ICommand
{
  public void Execute()
  {
    Console.WriteLine("Opening a file");
  }
}
```

Now, in your editor, you want to create a bunch of buttons. Each button, when pressed, executes the corresponding command. We can represent the button as follows:

```
public class Button
{
  private ICommand command;
  private string name;

  public Button(ICommand command, string name)
  {
    this.command = command;
    this.name = name;
  }

  public void Click() { command.Execute(); }
  public void PrintMe()
  {
    Console.WriteLine($"I am a button called {name}");
  }
}
```

Now, here is a challenge: how do you make an editor that has a single button created for each of the commands registered in the system? We can begin by defining it as follows:

```
public class Editor
{
  public IEnumerable<Button> Buttons { get; }

  public Editor(IEnumerable<Button> buttons)
  {
    Buttons = buttons;
  }
}
```

Now we can set up a dependency injection container with all the possible commands. We can also add a bit of metadata to each command, storing its name:

```
var b = new ContainerBuilder();
b.RegisterType<OpenCommand>()
  .As<ICommand>()
  .WithMetadata("Name", "Open");
b.RegisterType<SaveCommand>()
  .As<ICommand>()
  .WithMetadata("Name", "Save");
```

We can now register an Adapter inside the DI container that will construct a `Button` for each registered command and, furthermore, will take the metadata `Name` value from each command and pass it as the second constructor argument:

```
b.RegisterAdapter<Meta<ICommand>, Button>(cmd =>
        new Button(cmd.Value, (string)cmd.Metadata["Name"]));
```

We can now register the `Editor` itself and build the container. When we resolve the editor, its constructor will receive an `IEnumerable<Button>` with one button per registered command:

```
b.RegisterType<Editor>();

using var c = b.Build();
var editor = c.Resolve<Editor>();
foreach (var btn in editor.Buttons)
  btn.PrintMe();
// I am a button called Open
// I am a button called Save
```

So, as you can see, while this is not an adapter in the classic sense, it allows us to enforce 1-to-1 correspondance between a set of types conforming to some criteria and a set of instances related to those types.

Adapters in the .NET Framework

There are many uses of the Adapter pattern in .NET Framework, including:

- ADO.NET providers living in `System.Data`, such as `SqlCommand`, adapt an OOP-defined database command or query to be executed using SQL. Each ADO.NET provider is an adapter for a particular database type.
- Database data adapters – types that inherit from `DbDataAdapter` – perform a similar, higher-level operation. Internally, they represent a set of data commands and a connection to a particular data source (a database, typically), their goal being to populate a `DataSet` and update the data source.
- LINQ providers are also adapters, each adapting some underlying storage technology to be usable through LINQ operators (`Select`, `Where`, etc.). Expression trees exist with the central purpose of translating conventional C# lambda functions into other query languages and mechanisms such as SQL.

- Stream adapters (e.g., `TextReader`, `StreamWriter`) adapt a stream to read a particular type of data (binary, text) into a particular type of object. For example, a `StringWriter` writes into a buffer held by a `StringBuilder`.
- WPF uses the `IValueConverter` interface to allow scenarios such as a text field being bound to a numeric value. Unlike most of the adapters here, this one is *bidirectional*, meaning that the interface is adapted in both directions: changes to a numeric field/property get turned into text displayed in the control and, conversely, text entered into the control gets parsed and converted into a numeric value.
- Interop-related entities in C# represent the Adapter pattern. For example, that dummy P/Invoke type that you write allows you to adapt a C/C++ library to your C# needs. Same goes for Runtime-Callable Wrappers (RCWs), which allow managed classes and COM components to interact despite their obvious interface differences.

Summary

Adapter is a very simple concept: it allows you to adapt the interface you have to the interface you need. The only real issue with adapters is that, in the process of adaptation, you sometimes end up generating temporary data so as to satisfy requirements related to representation of data in a form palatable for the target API. And when this happens, we turn to caching: ensuring that new data is only generated when necessary. If we are implementing caching with a special key, we need to ensure that collisions are either impossible or are handled appropriately. If we are using the object itself as the underlying key, the presence of both `GetHashCode()` and `Equals()` solves this problem for us.

As an additional optimization, we can ensure that the adapter doesn't generate the temporaries immediately, but instead only generates them when they are actually required. Further optimizations are possible but are domain-specific: for example, in our case lines can be parts of other lines, which would let us to further save on the number of `Point` objects created.

Bridge

One very common situation that occurs when designing software is the so-called *state space explosion* where the number of related entities required to represent all possible states 'explodes' in a Cartesian product fashion. For example, if you have circles and squares of different colors, you might end up with classes such as `RedSquare`/`BlueSquare`/`RedCircle`/`BlueCircle` etc. Clearly nobody wants that.

What we do instead is we connect things together, and there are different ways of doing that. For example, if object color is simply a trait, we create an `enum`. But if color has mutable fields, properties or behaviors, we cannot restrict ourselves to an `enum`: if we do, we'll have plenty of `if/switch` statements in unrelated classes. Again, it's not something that we want.

The Bridge pattern essentially connects constituent aspects of an object using, ahem, references. Not very exciting, is it? Well, I can offer some excitement towards the end of our exploration, but we first need to take a look at a conventional implementation of the pattern.

Conventional Bridge

Let's imagine that we are interested in drawing different kinds of shapes on the screen. Let's suppose that we have a variety of shapes (circle, square, etc.) and also different APIs for rendering these (say, raster vs. vector rendering).

We want to create objects which specify both the type of shape as well as the rendering mechanism the shape should use for rendering. How can we do this? Well, on the one hand, we can define an infinite bunch of classes (`RasterSquare`, `VectorCircle` etc.) and provide an implementation for each. Or we could somehow get each shape to refer to which renderer it's using.

Let's begin by defining an `IRenderer`. This interface will determine how different shapes are rendered by whatever mechanism is required:[15]

```
public interface IRenderer
{
  void RenderCircle(float radius);
  // RenderSquare, RenderTriangle, etc.
}
```

And, on the other side, we can define an abstract class (not an interface) for our shape hierarchy. Why an abstract class? Because we want to keep a reference to the renderer.

```
public abstract class Shape
{
  protected IRenderer renderer;

  // a bridge between the shape that's being drawn an
  // the component which actually draws it
  public Shape(IRenderer renderer)
  {
    this.renderer = renderer;
  }

  public abstract void Draw();
  public abstract void Resize(float factor);
}
```

This may seem counterintuitive, so let's make a pause and ask ourselves: what are we trying to guard against, exactly? Well, we're trying to handle two situations: when new renderers get added, and when new shapes get added to the system. We don't want either of these to spawn *multiple* changes. So here are the two situations:

- If a new shape gets added, all it has to do is inherit `Shape` and implement its members (say there is M different ones). Each renderer then has to implement

[15] I am being sly here by using a calling convention. This is done purely for illustration purposes. If every rendered shape shares is neither parent nor child of another shape, you can simplify this by making a series of similarly-named overloads, i.e., `Render(Circle c)`, `Render(Square s)` and so on. The choice is up to you.

  just one new member (`RenderXxx`). So if there are M different renderers, the total number of operations required for a new shape is M+N.
- If a new renderer gets added, all it has to do is implement M different members, one for every shape.

As you can see, we either implement M members or M+N members. At no point do we get an M-by-N situation, which is what the pattern actively tries to avoid. Another added is that renderers always know how to render all the shapes available in the system because each shape `Xxx` has a `Draw()` method that explicitly calls `RenderXxx()`.

So here is the implementation of the `Circle`:

```
public class Circle : Shape
{
  private float radius;

  public Circle(IRenderer renderer, float radius) : base(renderer)
  {
    this.radius = radius;
  }

  public override void Draw()
  {
    renderer.RenderCircle(radius);
  }

  public override void Resize(float factor)
  {
    radius *= factor;
  }
}
```

And here is a sample implementation of one of the renderers:

```
public class VectorRenderer : IRenderer
{
  public void RenderCircle(float radius)
  {
    WriteLine($"Drawing a circle of radius {radius}");
  }
}
```

Notice that the `Draw()` method simply uses the bridge: it calls the corresponding renderer's drawing implementation for this particular object.

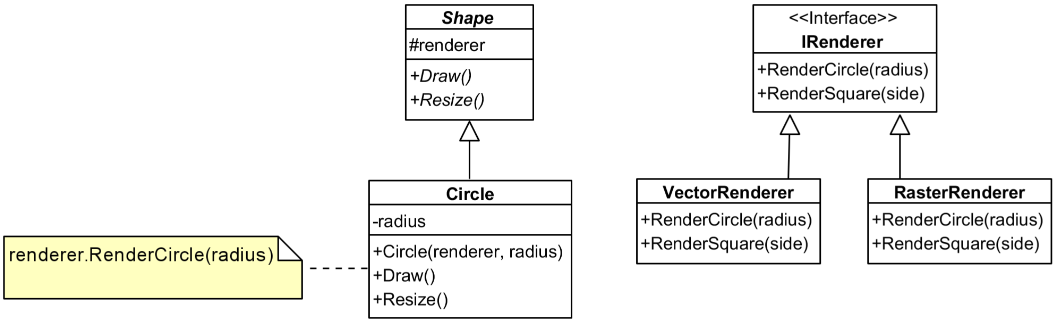
In order to use this set-up, you have to instantiate both an `IRenderer` as well as the shape. This can be done either directly:

```
var raster = new RasterRenderer();
var vector = new VectorRenderer();
var circle = new Circle(vector, 5);
circle.Draw(); // Drawing a circle of radius 5
circle.Resize(2);
circle.Draw(); // Drawing a circle of radius 10
```

Or, if you are using a dependency injection framework, you can define a default renderer to be used throughout the application. This way, all constructed instances of a `Circle` will be preinitialized with a renderer that is centrally defined. Here is an example that uses the Autofac container:

```
var cb = new ContainerBuilder();
cb.RegisterType<VectorRenderer>().As<IRenderer>();
cb.Register((c, p) => new Circle(c.Resolve<IRenderer>(),
  p.Positional<float>(0)));
using (var c = cb.Build())
{
  var circle = c.Resolve<Circle>(
    new PositionalParameter(0, 5.0f)
  );
  circle.Draw();
  circle.Resize(2);
  circle.Draw();
}
```

The above specifies that, by default, a `VectorRenderer` should be provided when someone asks for an `IRenderer`. Furthermore, since shapes take an additional parameter (their size, presumably), we specify a default value of zero.



Dynamic Prototyping Bridge

You may have noticed that the Bridge is nothing more than the application of the Dependency Inversion Principle, where you connect two distinct hierarchies together through a common parameter. So now we are going to take a look at a more sophisticated example involving something called Dynamic Prototyping.

Dynamic Prototyping is a technique for editing .NET programs *while they are running*. You have have already experienced this as the "Edit & Continue" feature in Visual Studio. The idea of dynamic prototyping is to allow the user to make immediate changes to the program that's currently running by editing and run-time-compiling the program's source code.

How does it work? Well, imagine you are sticking to 'one class per file' approach and you know in advance that your DI container can satisfy *all* dependencies of a given class. What you can do in this case is:

- Allow the user to edit the source code of this class. This works best if there is 1-to-1 correspondence between the class and the file. Most modern IDEs try to enforce this approach.
- After you edit and save the new source code, you use the C# compiler to complile just that class and get an in-memory implementation of the new type. You basically get a `System.Type`. You could, if you wanted, just instantiate that new type and use it to update some reference, or…

- You change the registration options in the DI container so that your new type is now a replacement for the original type. This naturally requires that you are using abstractions of some kind.

The last point needs explaining. If you have a concrete type `Foo.Bar` and you build a brand new, in-memory type `Foo.Bar` then, even if the APIs of those types stay the same, those types are *incompatible*. You cannot assign a reference to the old `Bar` with the new one. The only way to use them interchangeably is via `dynamic` or reflection, and both of those are niche cases.

Let me illustrate how the entire process works. Suppose you have an `Log` class being used by a `Payroll` class. Using hypothetical dependency injection, you could define it as:

```
// Log.cs
public class Log
{
  void Info(string msg) { ... }
}

// Payroll.cs
public class Payroll
{
  [Service]
  public Log Log { get; set; }
}
```

Notice that I'm defining the `Log` as an injected property, not via constructor injection. Now, to make a dynamic bridge, you would introduce an interface, i.e.

```
// ILog.cs
public interface ILog
{
  void Info(string msg);
}

// Log.cs
public class Log : ILog { /* as before */ }

// Payroll.cs
public class Payroll
{
  [Service]
  public ILog Log { get; set; }
}
```

Pay attention to the names of files, too. This is important, each type is in its own file. Now, as you run this program, suppose you want to change the implementation of Log without stopping the application. What you would do is:

- Open up an editor with the Log.cs file and edit the file.
- Close the editor. Now Log.cs gets compiled into an in-memory assembly.
- Create the first type found in this new assembly. It will be a Log, for sure, but incompatible with the previous Log! However, it implements an ILog, which is good enough for us.
- Go over the objects already created by the container and update all [Service]-marked references to an ILog with the new object.

This last part could be tricky. First, you need a container that can go over its own injection points though, to be honest, you could use good old-fashioned reflection for this purpose too. The reason I'm referring to a container is that it's more convenient to use. Also, notice this approach only works for property injection, and there's an implicit assumption that the service is *immutable* (has no state). If the service had state, you'd have to serialize it and then deserialize the data into the new object – not impossible, but a robust implementation needs to handle many corner cases.

So the moral of this story is that, in order to be able to substitute one runtime-constructed type for another, they both need to implement the same interface.

And before you ask, *no*, you cannot dynamically change any base type (class or interface).

Summary

The principal goal of the Bridge design pattern, as we have seen, is to avoid excessive proliferation of data types where there are two or more ‘dimensions’, i.e., aspects of a system that can potentially multiply in number. The best approach to Bridge is still active avoidance (e.g., replace classes with enums, if possible) but if that’s not possible, we simply abstract away both hierarchies and find a way of connecting them.

Composite

It's a fact of life that objects are quite often composed of other objects (or, in other words, they aggregate other objects). Remember, we agreed to equate aggregation and composition at the start of this part of the book.

There are a few ways for an object to advertise that it's composed of something. The most obvious approach is for an object to either implement `IEnumerable<T>` (where `T` is whatever you're prepared to expose) or, alternatively, to expose public members that themselves implement `IEnumerable<T>`.

Another option for advertising being a composite is to inherit from a known collection class such as `Collection<T>`, `List<T>` or similar. This of course lets you not just implement `IEnumerable<T>` implicitly, but also provides you with an internal storage mechanism, so issues such as adding new objects to the collection are automatically handled for you.

So, what is the Composite pattern about? Essentially, we try to give single objects and groups of objects an identical interface and have those interface members work correctly regardless of which class is the underlying.

Grouping Graphic Objects

Think of an application such as PowerPoint where you can select several different objects and drag them as one. And yet, if you were to select a single object, you can grab that object too. Same goes for rendering: you can render an individual graphic object, or you can group several shapes together and they get drawn as one group.

The implementation of this approach is rather easy because it relies on just a single base class such as the following:

```
public class GraphicObject
{
  public virtual string Name { get; set; } = "Group";
  public string Color;
  // todo members
}
public class Circle : GraphicObject
{
  public override string Name => "Circle";
}
public class Square : GraphicObject
{
  public override string Name => "Square";
}
```

This appears to be a fairly ordinary example with nothing standing out except for the fact that `GraphicObject` is abstract, as well as the `virtual string Name` property which is, for some reason, set to “Group”. So even though the inheritors of `GraphicObject` are, obviously, scalar entities, `GraphicObject` itself reserves the right to act as a *container* for further items.

The way this is done is by furnishing `GraphicObject` with a lazily-constructed list of children:

```
public class GraphicObject
{
  ...
  private readonly Lazy<List<GraphicObject>> children =
    new Lazy<List<GraphicObject>>();
  public List<GraphicObject> Children => children.Value;
}
```

So `GraphicObject` can act as both a singular, scalar element (you inherit it and you get a `Circle`, for example), but it can also be used as a container of elements. We can implement some methods that would print its contents:

```
public class GraphicObject
{
  private void Print(StringBuilder sb, int depth)
  {
    sb.Append(new string('*', depth))
      .Append(string.IsNullOrWhiteSpace(Color) ? string.Empty : $"{Color} ")
      .AppendLine($"{Name}");
    foreach (var child in Children)
      child.Print(sb, depth + 1);
  }

  public override string ToString()
  {
    var sb = new StringBuilder();
    Print(sb, 0);
    return sb.ToString();
  }
}
```

The above code uses asterisks for indicating the level of depth of each element. Armed with this, we can now construct a drawing that consists of both shapes as well as group of shapes and print it out:

```
var drawing = new GraphicObject {Name = "My Drawing"};
drawing.Children.Add(new Square {Color = "Red"});
drawing.Children.Add(new Circle{Color="Yellow"});

var group = new GraphicObject();
group.Children.Add(new Circle{Color="Blue"});
group.Children.Add(new Square{Color="Blue"});
drawing.Children.Add(group);

WriteLine(drawing);
```

And here is the output we get:

```
My Drawing
*Red Square
*Yellow Circle
*Group
**Blue Circle
**Blue Square
```

So this is the simplest implementation of the Composite design pattern that is based on inheritance and optional containment of a list of sub-elements. The only issue, which the astute reader will point out, is that it makes absolutely no sense for scalar classes such as `Circle` or `Square` to have a `Children` member. What if someone were to use such an API? It would make very little sense.

In the next example we're going to look at scalar objects which are truly scalar, with no extraneous members in their interface.

Neural Networks

Machine Learning is the hot new thing, and I hope it stays this way, or I'll have to update this paragraph. Part of machine learning is the use of artificial neural networks: software constructs which attempt to mimic the way neurons work in our brains.

The central concept of neural networks is, of course, a *neuron*. A neuron can produce a (typically numeric) output as a function of its inputs, and we can feed that value on to other connections in the network. We're going to concern ourselves with connections only, so we'll model the neuron like so:

```
public class Neuron
{
  public List<Neuron> In, Out;
}
```

This is a simple neuron with outgoing and incoming connections to other neurons. What you probably want to do is to be able to connect one neuron to another, which can be done using

```
public void ConnectTo(Neuron other)
{
  Out.Add(other);
  other.In.Add(this);
}
```

This method does fairly predictable things: it sets up connections between the current (this) neuron and some other one. So far so good.

Now, suppose we also want to create neuron *layers*. A layer is quite simply a specific number of neurons grouped together. This can easily be done just by inheriting from a Collection<T>, i.e.:

```
public class NeuronLayer : Collection<Neuron>
{
  public NeuronLayer(int count)
  {
    while (count --> 0)
      Add(new Neuron());
  }
}
```

Looks good, right? I've even thrown in the arrow --> operator for you to enjoy.[16] But now, we've got a bit of a problem.

The problem is this: we want to be able to have neurons connectable to neuron layers (in both directions), and we also want layers to be connectable to other layers. Broadly speaking, we want this to work:

[16]There is, of course, no --> operator; it's quite simply the postfix decrement -- followed by greater-than >. The effect, though, is exactly as the --> arrow suggests: in while (count --> 0) we iterate until count reaches zero.

```
var neuron1 = new Neuron();
var neuron2 = new Neuron();
var layer1 = new NeuronLayer(3);
var layer2 = new NeuronLayer(4);

neuron1.ConnectTo(neuron2); // works already :)
neuron1.ConnectTo(layer1);
layer2.ConnectTo(neuron1);
layer1.ConnectTo(layer2);
```

As you can see, we've got 4 distinct cases to take care of:

1. Neuron connecting to another neuron
2. Neuron connecting to layer
3. Layer connecting to neuron; and
4. Layer connecting to another layer

As you may have guessed, there's no way in Baator that we'll be making 4 overloads of the `ConnectTo()` method. What if there were 3 distinct classes – would we realistically consider creating 9 methods? I do not think so.

The way to have a single-method solution to this problem is to realize that both `Neuron` and `NeuronLayer` can be treated as enumerables. In the case of a `NeuronLayer`, there's no problem – it is already enumerable, but in the case of `Neuron`, well… we need to do some work.

In order to get `Neuron` ready, we are going to:

- Remove its own `ConnectTo()` method, since it's not general enough.
- Implement the `IEnumerable<Neuron>` interface, yielding… ourself (!) when someone wants to enumerate us.

Here's what the new `Neuron` class looks like:

```
public class Neuron : IEnumerable<Neuron>
{
  public List<Neuron> In, Out;

  public IEnumerator<Neuron> GetEnumerator()
  {
    yield return this;
  }

  IEnumerator IEnumerable.GetEnumerator()
  {
    return GetEnumerator();
  }
}
```

And now, the *piece de resistance*: since both `Neuron` and `NeuronLayer` now conform to `IEnumerable<Neuron>`, all that remains for us to do is to implement a single extension method that connects the two enumerables together:

```
public static class ExtensionMethods
{
  public static void ConnectTo(
    this IEnumerable<Neuron> self, IEnumerable<Neuron> other)
  {
    if (ReferenceEquals(self, other)) return;

    foreach (var from in self)
      foreach (var to in other)
      {
        from.Out.Add(to);
        to.In.Add(from);
      }
  }
}
```

And that's it! We now have a single method that can be called to glue together any entities consisting of `Neuron` classes. Now, if we decided to make some `NeuronRing`, provided it supports `IEnumerable<Neuron>`, we can easily connect it to either a `Neuron`, a `NeuronLayer` or another `NeuronRing`!

Shrink Wrapping the Composite

No doubt many of you want some kind of prepackaged solution that would allow scalar objects to be treated as enumerables. Well, if your scalar class doesn't derive from another class, you can simply define a base class similar to the following:

```
public abstract class Scalar<T> : IEnumerable<T>
  where T : Scalar<T>
{
  public IEnumerator<T> GetEnumerator()
  {
    yield return (T) this;
  }

  IEnumerator IEnumerable.GetEnumerator()
  {
    return GetEnumerator();
  }
}
```

This class is generic, and the type parameter T refers to the object we're trying to 'scalarize'. Now, making any object expose itself as a collection of one element is as simple as:

```
public class Foo : Scalar<Foo> {}
```

and the object is immediately available for use in, say, a `foreach` loop:

```
var foo = new Foo();
foreach (var x in foo)
{
  // will yield only one value of x
  // where x == foo referentially :)
}
```

This approach only works if your type doesÑ,'t have a parent because multiple inheritance is impossible. It would, of course, be nice to have some marker interface (inheriting from IEnumerable<T>, perhaps, though that's not strictly necessary) that would implement GetEnumerator() as extension methods. Sadly, the C# language designers did not leave this option available – GetEnumerator() must strictly be *instance* methods to be picked up by foreach.

Regrettably, we cannot abuse C# 8's default interface members to use an interface instead of a class to shrink-wrap the composite. The reason for this is that you must explicitly cast the class to the interface that contains default members, so if you're hoping for GetEnumerator() duck typing, you're out of luck.

The best we can come up with is something like this:

```
public interface IScalar<out T>
  where T : IScalar<T>
{
  public IEnumerator<T> GetEnumerator()
  {
    yield return (T) this;
  }
}
```

Notice that this interface *cannot* inherit from IEnumerable<T>. Well, you *can* inherit it, but it won't save you from having to implement GetEnumerator() pairs in the class *anyway*, which completely defeats the point.

What can you do with the above interface? Well, you could use it in a class:

```
public class Foo : IScalar<Foo> { ... }
```

But unfortunately when it comes to iterating the thing, you would have to perform a cast before duck typing does its work:

```
var foo = new Foo();
var scalar = foo as IScalar<Foo>; // :(
foreach (var f in scalar)
{
  ...
}
```

Of course, if we could trick the system with this, we could have equally tricked it years ago by defining extension methods and marker interfaces for scalars. Sadly, we're out of luck here.

Composite Specification

When I introduced the Open-Closed Principle, I gave a demo of the Specification pattern. The key aspects of the pattern were base types `IFilter` and `ISpecification` that allowed us to use inheritance to build an extensible filtering framework that conformed to the OCP. Part of that implementation involved combinators – specifications what would combine several specifications together under an AND or OR operator mechanic.

Both `AndSpecification` and `OrSpecification` made use of two operands (which we called `left` and `right`), but that restriction was completely arbitrary: in fact, we could have combined more than two elements together and, furthermore, we could improve the OOP model with a reusable base class such as the following:

```
public abstract class CompositeSpecification<T> : ISpecification<T>
{
  protected readonly ISpecification<T>[] items;

  public CompositeSpecification(params ISpecification<T>[] items)
  {
    this.items = items;
  }
}
```

The above should feel familiar because we've implemented the approach before.

We made an `ISpecification` that is, in fact, a combination of different specifications passed as `params` in the constructor.

With this approach, the `AndSpecification` combinator can now be implemented with a bit of LINQ:

```
public class AndSpecification<T> : CompositeSpecification<T>
{
  public AndSpecification(params ISpecification<T>[] items) : base(items)
  {
  }

  public override bool IsSatisfied(T t)
  {
    return items.All(i => i.IsSatisfied(t));
  }
}
```

Similarly, if you wanted an `OrSpecification`, you would replace the call to `All()` with a call to `Any()`. You could even make specifications that would support other, more complicated criteria. For example, you could make a composite such that the item is required to satisfy at most/at least/specifically a number of specifications contained within.

Summary

The Composite design pattern allows us to provide identical interfaces for individual objects and collections of objects. This can be accomplished in one of two ways:

- Make every scalar object you intend to work with a collection or, alternatively, get it to contain a collection and expose it somehow. You can use `Lazy<T>` so you don't allocate too many data structures if they are not actually needed. This is a very simple approach and is somewhat unidiomatic.
- Teach scalar objects to appear as collections. This is done by implementing `IEnumerable<T>` and then calling `yield return this` in `GetEnumerator()`. Strictly speaking, having a scalar value expose `IEnumerable` is also

unidiomatic, but it is aesthetically better and has a smaller computational cost.

Decorator

Suppose you're working with a class your colleague wrote, and you want to extend that class' functionality. How would you do it, without modifying the original code? Well, one approach is inheritance: you make a derived class, add the functionality you need, maybe even `override` something, and you're good to go.

Right, except this doesn't always work, and there are many reasons why. The most common reason is that you cannot inherit the class – either because your target class needs to inherit something else (and multiple inheritance is impossible) or because the class you want to extend is `sealed`.

The Decorator pattern allows us to enhance existing types without either modifying the original types (Open-Closed Principle) or causing an explosion of the number of derived types.

Custom String Builder

Suppose you're into code generation and you want to extend `StringBuilder` in order to offer additional utility methods such as supporting indentation or scopes or whatever code generation functionality makes sense. It would be nice to simply inherit from `StringBuilder`, but it's `sealed` (for security reasons). Also, since you might want to store the current indentation level (say, to provide `Indent()`/`Unindent()` methods), you cannot simply go ahead and use extension methods, since those are stateless.[17]

So the solution is to create a Decorator: a brand new class that aggregates a `StringBuilder` but also stores and exposes the same members as StringBuilder did, and a few more. From the outset, the class can look as follows:

[17]Strictly speaking, it *is* possible to store state in extension methods, albeit in a very roundabout way. Essentially, what you'd do is have your extension class keep a `static member` of type `Dictionary<WeakReference, Dictionary<string,object>>` and then modify the entries in this dictionary to map an object to its set of properties. Plenty of fiddling is required here, both in terms of working with weak references (we don't want this store to extend the lifetime of the original object, right?) as well as the boxing and unboxing that comes with storing a bunch of `objects`.

```
public class CodeBuilder
{
  private StringBuilder builder = new StringBuilder();
  private int indentLevel = 0;

  public CodeBuilder Indent()
  {
    indentLevel++;
    return this;
  }
}
```

As you can see, we have both the ‘underlying’ `StringBuilder` as well as some additional members related to the extended functionality. What we need to do now is expose the members of `StringBuilder` as members of `CodeBuilder`, delegating the calls. `StringBuilder` has a very large API, so doing this by hand is unreasonable: instead, what you would do is use code generation (e.g., ReSharper’s **Generate | Delegated members**) to automatically create the necessary API.

This operation can be applied to every single member of `StringBuilder` and will generate the following signatures:

```
public class CodeBuilder
{
  public StringBuilder Append(string value)
  {
    return builder.Append(value);
  }

  public StringBuilder AppendLine()
  {
    return builder.AppendLine();
  }
  // other generated members omitted
}
```

This might seem great on first glance, but in actual fact, the implementation is incorrect. Remember, `StringBuilder` exposes a fluent API in order to be able to write things like

```
myBuilder.Append("Hello").AppendLine(" World");
```

In other words, it provides a fluent interface. But our decorator does not! For example, it won't let us write `myBuilder.Append("x").Indent()` because the result of `Append()`, as generated by ReSharper, is a `StringBuilder` which doesn't have an `Indent()` member. That's right – ReSharper does not know that we want a proper fluent interface. What you want is the fluent calls in `CodeBuilder` to appear as:

```
public class CodeBuilder
{
  public CodeBuilder Append(char value, int repeatCount)
  {
    builder.Append(value, repeatCount);
    return this; // return a CodeBuilder, not a StringBuilder
  }
  ...
}
```

This is something that you'd need to fix by hand or possibly through regular expressions. This modification, when applied to every single call that's delegated to `StringBuilder`, would allow us to chain `StringBuilder`'s calls together with our unique, `CodeBuilder`-specific ones.

Adapter-Decorator

You can also have a decorator that acts as an adapter. For example, suppose we want to take the `CodeBuilder` from the above, but we want it to start acting as a `string`. Perhaps we want to take a `CodeBuilder` and stick it into an API which expects our object to implement the = operator for assigning from a `string` and a += operator for appending additional strings. Can we adapt `CodeBuilder` to these requirements? We sure can; all we have to do is add the appropriate functionality:

```
public static implicit operator CodeBuilder(string s)
{
  var cb = new CodeBuilder();
  cb.sb.Append(s);
  return cb;
}

public static CodeBuilder operator +(CodeBuilder cb, string s)
{
  cb.Append(s);
  return cb;
}
```

With this implementation, we can now start working with a `CodeBuilder` as if it were a `string`:

```
CodeBuilder cb = "hello";
cb += " world";
WriteLine(cb); // prints "hello world"
```

Curiously enough, the second line in the above will work even if we didn't implement `operator +` explicitly. Why? You figure it out!

Multiple Inheritance with Interfaces

In addition to extending `sealed` classes, the Decorator also shows up when you want to have multiple base classes… which of course you cannot have because C# does not support multiple inheritance. For example, suppose you have a Dragon that's both a Bird and a Lizard. It would make sense to write something like:

```
public class Bird
{
  public void Fly() { ... }
}

public class Lizard
{
  public void Crawl() { ... }
}

public class Dragon : Bird, Lizard {} // cannot do this!
```

Sadly, this is impossible, so what do you do? Well, you extract interfaces from both `Bird` and `Lizard`

```
public interface IBird
{
  void Fly();
}

public interface ILizard
{
  void Crawl();
}
```

Then you make a `Dragon` class that implements these interfaces, aggregates instances of `Bird` and `Lizard` and delegates the calls:

```
public class Dragon: IBird, ILizard
{
  private readonly IBird bird;
  private readonly ILizard lizard;

  public Dragon(IBird bird, ILizard lizard)
  {
    this.bird = bird;
    this.lizard = lizard;
  }
  public void Crawl()
  {
    lizard.Crawl();
  }

  public void Fly()
  {
    bird.Fly();
  }
}
```

You'll notice that there are two options here: either you initialize the default instances of `Bird` and `Lizard` right inside the class or you offer the client more flexibility by taking both of those objects in the constructor. This would allow you to construct more sophisticated `IBird`/`ILizard` classes and make a dragon out of them. Also, this approach automatically supports constructor injection, should you go the IoC route.

One interesting problem with the decorator is the 'diamond inheritance' problem of C++. Suppose a dragon crawls only until it's 10 years old and, from then on, it only flies. In this case, you'd have both the `Bird` and `Lizard` classes have an Age property with independent implementations:

```
public interface ICreature
{
  int Age { get; set; }
}

public interface IBird : ICreature
{
  void Fly();
}

public interface ILizard : ICreature
{
  void Crawl();
}
```

```
public class Bird : IBird
{
  public int Age { get; set; }
  public void Fly()
  {
    if (Age >= 10)
      WriteLine("I am flying!");
  }
}

public class Lizard : ILizard
{
  public int Age { get; set; }
  public void Crawl()
  {
    if (Age < 10)
      WriteLine("I am crawling!");
  }
}
```

Notice that we've had to introduce a new interface `ICreature` just so we could expose the `Age` as part of both the `IBird` and `ILizard` interfaces. The real problem here is the implementation of the `Dragon` class, because if you use the code generation features of ReSharer or similar tool, you will simply get:

```
public class Dragon : IBird, ILizard
{
  ...
  public int Age { get; set; }
}
```

This once again shows that generated code isn't always what you want. Remember, both `Bird.Fly()` and `Lizard.Crawl()` have *their own* implementations of `Age`, and those implementations need to be kept consistent in order for those methods to operate correctly. This means that the correct implementation of `Dragon.Age` is the following:

```
public int Age
{
  get => bird.Age;
  set => bird.Age = lizard.Age = value;
}
```

Notice that our setter assigns both, whereas the getter simply uses the underlying `bird` – this choice is arbitrary, we could have easily taken the `lizard`'s age instead. The setter ensures consistency, so in theory, both values would always be equal… except during initialization, a place we haven't taken care of yet. A lazy man's solution to this problem would be to redefine the `Dragon` constructor thus:

```
public Dragon(IBird bird, ILizard lizard)
{
  this.bird = bird;
  this.lizard = lizard;
  bird.Age = lizard.Age;
}
```

As you can see, building a decorator is generally easy, except for two nuances: the difficulties in preserving a fluent interface, and the challenge of diamond inheritance. I have demonstrated here how to solve both of these problems.

Multiple Inheritance with Default Interface Members

The collision between the `Age` properties of `Bird` and `Lizard` can be partially mitigated with C# 8's default interface members. While they do not give us 'proper', C++ style multiple inheritance, they give us enough to go by.

First of all, we implement a base interface for a creature:

```
public interface ICreature
{
  int Age { get; set; }
}
```

This step is essential, because now we can define interfaces `IBird` and `ILizard` that have default method implementations that actually make use of the property:

```
public interface IBird : ICreature
{
  void Fly()
  {
    if (Age >= 10)
      WriteLine("I am flying");
  }
}

public interface ILizard : ICreature
{
  void Crawl()
  {
    if (Age < 10)
      WriteLine("I am crawling!");
  }
}
```

Finally, we can make a class that implements both of these interfaces. Of course, this class has to provide an implementation of the Age property, since no interface is able to do so:

```
public class Dragon : IBird, ILizard
{
  public int Age { get; set; }
}
```

And now we have a class that inherits the behavior of two interfaces. The only caveat is that explicit casts are required to actually make use of those behaviors:

```
var d = new Dragon {Age = 5};

if (d is IBird bird)
  bird.Fly();

if (d is ILizard lizard)
  lizard.Crawl();
```

Dynamic Decorator Composition

Of course, as soon as we start building decorators over existing types, we come to the question of decorator *composition*, i.e., whether or not it's possible to decorate a decorator with another decorator. I certainly hope it's possible – decorators should be flexible enough to do this!

For our scenario, let's imagine that we have an abstract base class called `Shape` with a single member called `AsString()` that returns a string describing this shape (I'm deliberately avoiding `ToString()` here):

```
public abstract class Shape
{
  public virtual string AsString() => string.Empty;
}
```

I chose to make `Shape` an abstract class with a default, no-op implementation. We could equally use an `IShape` interface for this example.

We can now define a concrete shape like, say, a circle or a square:

```
public sealed class Circle : Shape
{
  private float radius;

  public Circle() : this(0)
  {

  }

  public Circle(float radius)
  {
    this.radius = radius;
  }

  public void Resize(float factor)
  {
    radius *= factor;
  }

  public override string AsString() => $"A circle of radius {radius}";
}

// similar implementation of Square with 'side' member omitted
```

I deliberately made `Circle` and similar classes `sealed` so we cannot go ahead and simply inherit from them. Instead, we are once again going to build decorators: this time, we'll build two of them – one for adding color to a shape…

```
public class ColoredShape : Shape
{
  private readonly Shape shape;
  private readonly string color;

  public ColoredShape(Shape shape, string color)
  {
    this.shape = shape;
    this.color = color;
  }
  public override string AsString()
    => $"{shape.AsString()} has the color {color}";
}
```

and another to give a shape transparency:

```
public class TransparentShape : Shape
{
  private readonly Shape shape;
  private readonly float transparency;

  public TransparentShape(Shape shape, float transparency)
  {
    this.shape = shape;
    this.transparency = transparency;
  }

  public override string AsString() =>
    $"{shape.AsString()} has {transparency * 100.0f}% transparency";
}
```

As you can see, both of these decorators inherit from the abstract Shape class, so they are themselves Shapes and they decorate other Shapes by taking them in the constructor. This allows us to use them together, for example:

```
var circle = new Circle(2);
WriteLine(circle.AsString());
// A circle of radius 2

var redSquare = new ColoredShape(circle, "red");
WriteLine(redSquare.AsString());
// A circle of radius 2 has the color red

var redHalfTransparentSquare = new TransparentShape(redSquare, 0.5f);
WriteLine(redHalfTransparentSquare.AsString());
// A circle of radius 2 has the color red has 50% transparency
```

As you can see, the decorators can be applied to other Shapes in any order you wish, preserving consistent output of the AsString() method. One thing they do not guard against is cyclic repetition: you can construct a ColoredShape(ColoredShape(Square)) and the system will not complain; we could not detect this situation either, even if we wanted to.

So this is the *dynamic* decorator implementation: the reason why we call it dynamic is because these decorators can be constructed at runtime, objects wrapping objects as layers of an onion. It is, on the one hand, very convenient, but on the other hand, you lose all type information as you decorate the object. For example, a decorated Circle no longer has access to its Resize() member:

```
var redSquare = new ColoredShape(circle, "red");
redCircle.Resize(2); // oops!
```

This problem is impossible to solve: since ColoredShape takes a Shape, the only way to allow resizing is to add Resize() to Shape itself, but this operation might not make sense for all shapes. This is a limitation of the dynamic decorator.

Static Decorator Composition

When you are given a dynamically decorated ColorShape, there's no way to tell whether this shape is a circle, square or something else without looking at the output of AsString(). So how would you 'bake in' the underlying type of the decorated objects into the type of the object you have? Turns out you can do so with generics.

The idea is simple: our decorator, say ColoredShape, takes a generic argument that specifies what type of object it's decorating. Naturally, that object has to be a Shape and, since we're aggregating it, it's also going to need a constructor:

```
public class ColoredShape<T> : Shape
  where T : Shape, new()
{
  private readonly string color;
  private readonly T shape = new T();

  public ColoredShape() : this("black") {}
  public ColoredShape(string color) { this.color = color; }

  public override string AsString() =>
    return $"{shape.AsString()} has the color {color}";
}
```

Okay, so what's going on here? We have a new `ColoredShape` that's generic, it takes a `T` that's supposed to inherit a `Shape`. Internally, it stores an instance of `T` as well as color information. We've provided two constructors for flexibility: since C#, unlike C++, doesn't support constructor forwarding, the default constructor is going to be useful for composition (see, we have the `new()` requirement).

We can now provide a similar implementation of `TransparentShape<T>` and, armed with both, we can now build static decorators of the following form:

```
var blueCircle = new ColoredShape<Circle>("blue");
WriteLine(blueCircle.AsString());
// A circle of radius 0 has the color blue

var blackHalfSquare = new TransparentShape<ColoredShape<Square>>(0.4f);
WriteLine(blackHalfSquare.AsString());
// A square with side 0 has the color black has transparency 40
```

This static approach has certain advantages and disadvantages. The advantage is that we preserve the type information: given a `Shape` we can tell that the shape is a `ColoredShape<Circle>` and perhaps we can act on this information somehow. Sadly, this approach has plenty of disadvantages:

- Notice how the radius/side values in the above example are both zero. This is because we cannot initialize those values in the constructor: C# does not have constructor forwarding.

- We still don't have access to the underlying members; for example, `blue-Circle.Resize()` is still not legal.
- These sorts of decorators cannot be composed at runtime.

All in all, in the absence of CRTP[18] and mixin inheritance[19], the uses for static decorators in C# are very, very limited.

Functional Decorator

A functional decorator is a natural consequence of functional composition. If we can compose functions, we can equally wrap functions with other functions in order, for example, to provide before-and-after functionality such as logging.

Here's a very simple implementation. Imagine you have some work that needs to be done…

```
let doWork() =
  printfn "Doing some work"
```

We can now create a decorator function (a functional decorator!) that, given any function, measures how long it takes to execute:

```
let logger work name =
  let sw = Stopwatch.StartNew()
  printfn "%s %s" "Entering method" name
  work()
  sw.Stop()
  printfn "Exiting method %s; %fs elapsed" name sw.Elapsed.TotalSeconds
```

We can now use this wrapper around `doWork`, replacing a `unit -> unit` function with one with the same interface but which also performs some measurements:

[18]CRTP stands for Curiously Recurring Template Pattern; it is a popular C++ pattern which looks like this: `class Foo<T> : T`. In other words, you inherit from a generic parameter, something that's impossible in C#.

[19]Mixin inheritance is a C++ technique for adding functionality to classes by using inheritance. In the context of the decorator, it would allow us to compose a class of type `T<U<V>>` that would inherit from both `U` and `V`, giving us access to all the underlying members. Also, constructors would work correctly thanks to constructor forwarding and C++'s variadic templates.

```
let loggedWork() = logger doWork "doWork"
loggedWork()
// Entering method doWork
// Doing some work
// Exiting method doWork; 0.097824s elapsed
```

Pay attention to the round brackets in this example: it might be tempting to remove them, but that would drastically alter the types of data structures. Remember, any `let x = ...` construct will always evaluate to a variable (possibly of a `unit` type!) instead of a parameterless function unless you add an empty argument list.

There are a couple of catches in this implementation. For one, `doWork` does not return a value; if it did, we would have to cache it in a type-independent manner, something that's possible to implement in C++ but extremely difficult difficult to do in any .NET language. Another issue is that we have no way of determining the name of the wrapped function, so we end up passing it as a separate argument â€" not an ideal solution!

Summary

A decorator gives a class additional functionality while adhering to the OCP and mitigating issues related `sealed` classes and multiple inheritance. Its crucial aspect is *composability*: several decorators can be applied to an object in any order. We've looked at the following types of decorators:

- *Dynamic decorators* which can store references to the decorated objects and provide dynamic (runtime) composability.
- *Static decorators* which preserve the information about the type of the objects involved in the decoration; these are of limited use since they do not expose underlying objects' members, nor do they allow us to efficiently compose constructor calls.

In both of the cases, we completely ignored the issues related to cyclic use: nothing in the API prevents applying the same static or dynamic decorator more than once.

Façade

First of all, let's get the linguistic issue out of the way: that little curve in the letter Ç is called a *cedilla* and the letter itself is pronounced as an S, so the word 'façade' is pronounced as *fah-saad*. The particularly pedantic among you are welcome to use the letter ç in your code, since compilers treat this just fine.[20]

Now, about the pattern itself… essentially, the best analogy I can think of is a typical house. When you buy a house, you generally care about the exterior and the interior. You are less concerned about the internals: electrical systems, insulation, sanitation, that sort of thing. Those parts are all equally important, but we want them to 'just work' without breaking. You're much more likely to be buying new furniture than changing the wiring of your boiler.

The same idea applies to software: sometimes you need to interact with a complicated system in a simple way. By 'system' we could mean a set of components or just a single component with a rather complicated API. For example, think about the seemingly simple task of downloading a string of text from a URL. The fully fleshed-out solution to this problem using various `System.Net` data types looks like something like the following:

```
string url = "http://www.google.com/robots.txt";
var request = WebRequest.Create(url);
request.Credentials = CredentialCache.DefaultCredentials;
var response = request.GetResponse();
var dataStream = response.GetResponseStream();
var reader = new StreamReader(dataStream);
string responseFromServer = reader.ReadToEnd();
Console.WriteLine(responseFromServer);
reader.Close();
response.Close();
```

[20]Over the years I have seen many tricks involving the use of Unicode (typically UTF-8) encoding in C# source files. The most insidious case is one where a developer insisted on calling his extension methods' first argument `this` – it was, of course, a completely valid identifier because the letter `i` in `this` was a Ukrainian letter `i`, not a Latin one.

This is a lot of work! Furthermore, I almost guarantee that most of you wouldn't be able to write this code without looking it up on MSDN. That's because there are several underlying data types that make the operation possible. And if you wanted to do it all asynchronously, you would have to use a complementary API set comprised of `XxxAsync()` methods.

So whenever we encounter a situation where a complex interaction of different parts is required for something to get done, we might want to put it behind a façade, i.e., a much simpler interface. In the case of downloading web pages, all of the above reduces to a single line:

```
new WebClient().DownloadString(url);
```

In this example, the `WebClient` class is the façade, i.e., a nice, user-friendly interface that does what you want quickly and without ceremony. Of course, the original APIs are also available to you so that, if you need something more complicated (e.g., to provide credentials), you can use the more technical parts to fine-tune the operation of your program.

With this one example, you've already grasped the gist of the Façade design pattern. However, just to illustrate the matter further (as well as tell the story of how OOP is used and abused in practice), I would like to present yet another example.

Magic Squares

While a proper Facade demo requires that we make super-complicated systems that actually warrant a Facade to be put in front of them, let us consider a trivialized example: the process of making magic squares. A magic square is a matrix such as

```
  1 | 14 | 14 |  4
----+----+----+----
 11 |  8 |  6 |  9
----+----+----+----
  8 | 10 | 10 |  5
----+----+----+----
 13 |  2 |  3 | 15
```

If you add up the values in any row, any column, or any diagonal, you'll get the same number – in this case, 33. If we want to generate our own magic squares, we can imagine it as an interplay of three different subsystems:

- `Generator` – a component which simply generates a sequence of random numbers of a particular size.
- `Splitter` – a component that takes a rectangular matrix and outputs a set of lists representing all rows, columns and diagonals in the matrix.
- `Verifier` – a component that checks that the sums of all lists passed into it are the same.

We begin by implementing the `Generator`:

```
public class Generator
{
  private static readonly Random random = new Random();

  public List<int> Generate(int count)
  {
    return Enumerable.Range(0, count)
      .Select(_ => random.Next(1, 6))
      .ToList();
  }
}
```

Note that the generator yields 1-dimensional lists, whereas the next component, `Splitter`, accepts a matrix:

```
public class Splitter
{
  public List<List<int>> Split(List<List<int>> array)
  {
    // implementation omitted
  }
}
```

The implementation of `Splitter` is rather long-winded, so I've omitted it here – take a look at the source code for its exact details. As you can see, the `Splitter` returns a list-of-lists. Our final component, `Verifier`, checks that those lists all add up to the same number:

```
public class Verifier
{
  public bool Verify(List<List<int>> array)
  {
    if (!array.Any()) return false;

    var expected = array.First().Sum();

    return array.All(t => t.Sum() == expected);
  }
}
```

So there you have it – we have three different subsystems that are expected to work in concert in order to generate random magic squares. But are they easy to use? If we gave these classes to a client, they would really struggle to operate them correctly. So, how can we make their lives better?

The answer is simple: we build a Facade, essentially a wrapper class that hides all these implementation details and provides a very simple interface. Of course, it uses all the three classes behind the scenes:

```
public class MagicSquareGenerator
{
  public List<List<int>> Generate(int size)
  {
    var g = new Generator();
    var s = new Splitter();
    var v = new Verifier();

    var square = new List<List<int>>();

    do
    {
      square = new List<List<int>>();
      for (int i = 0; i < size; ++i)
        square.Add(g.Generate(size));
    } while (!v.Verify(s.Split(square)));

    return square;
  }
}
```

And there you have it! Now, if the client wants to generate a 3x3 magic square, all they have to do is call

```
var gen = new MagicSquareGenerator();
var square = gen.Generate(3);
```

And they'll get something like:

```
3 1 5
5 3 1
1 5 3
```

Okay, so this *is* a magic square, but maybe the user of this API has an additional requirement: they do not want numbers to repeat. How can we make it easy for them to implement this? First of all, we change `Generate()` to take each of the subsystems as generic parameters:

```
private List<List<int>> generate
  <TGenerator, TSplitter, TVerifier>(int size)
  where TGenerator : Generator, new()
  where TSplitter : Splitter, new()
  where TVerifier : Verifier, new()
{
  var g = new TGenerator();
  var s = new TSplitter();
  var v = new TVerifier();

  // rest of code as before
}
```

And now we simply make an overloaded `Generate()` that applies all three default generic parameters:

```
public List<List<int>> Generate(int size)
{
  return Generate<Generator, Splitter, Verifier>(size);
}
```

In the absence of default generic parameters, this is the only way we can provide sensible defaults and, at the same time, allow customization. Now, if the user wants to ensure all the values are unique, they can make a `UniqueGenerator`:

```
public class UniqueGenerator : Generator
{
  public override List<int> Generate(int count)
  {
    List<int> result;
    do
    {
      result = base.Generate(count);
    } while (result.Distinct().Count() != result.Count);

    return result;
  }
}
```

And then feed it into the Facade, thereby getting a better magic Square:

```
var gen = new MagicSquareGenerator();
var square = gen
  .Generate<UniqueGenerator, Splitter, Verifier>(3);
```

This gives us

```
8 1 6
3 5 7
4 9 2
```

Of course, it's really impractical to generate magic squares this way, but what this example demonstrates is that you can hide complicated interactions between different systems behind a Facade, and that you can also incorporate a certain amount of configurability so that users can customize the internal operations of the mechanism should the need arise.

Building a Trading Terminal

I've spent a lot of time working in areas of quant finance and algorithmic trading. As you can probably guess, what's required of a good trading terminal is quick delivery of information into a trader's brain: you want things to be rendered as fast as possible, without any lag.

Most of financial data (except for the charts) is actually rendered in plain text: white characters on a black screen. This is, in a way, similar to the way the terminal/console/command-line interface works in your own operating system, but there is a subtle difference.

The first part of a terminal window is the *buffer*. This is where the rendered characters are stored. A buffer is a rectangular area of memory, typically a 1D[21]

[21] Most buffers are typically one-dimensional. The reason for this is that it's easier to pass a single pointer somewhere than a double pointer, and using an `array` or `vector` doesn't make much sense when the size of the structure is deterministic and immutable. Another advantage to the 1D approach is that, when it comes to GPU processing, a system such as CUDA uses up to 6 dimensions for addressing *anyway*, so after a while, computing an 1D index from an N-dimensional block/grid position becomes second nature.

or 2D `char` or `wchar_t` array. A buffer can be much larger than the visible area of the terminal window, so it can store some historical output that you can scroll back to.

Typically, a buffer has a pointer (e.g., an integer) specifying the current input line. That way, a full buffer doesn't reallocate all lines; it just overwrites the oldest one.

Then there's the idea of a *viewport*. A viewport renders a part of the particular buffer. A buffer can be huge, so a viewport just takes a rectangular area out of that buffer and renders that. Naturally, the size of the viewport has to be less than or equal to the size of the buffer.

Finally, there's the console (terminal window) itself. The console shows the viewport, allows scrolling up and down, and even accepts user input. The console is, in fact, a façade: a simplified representation of what is a rather complicated set-up behind the scenes.

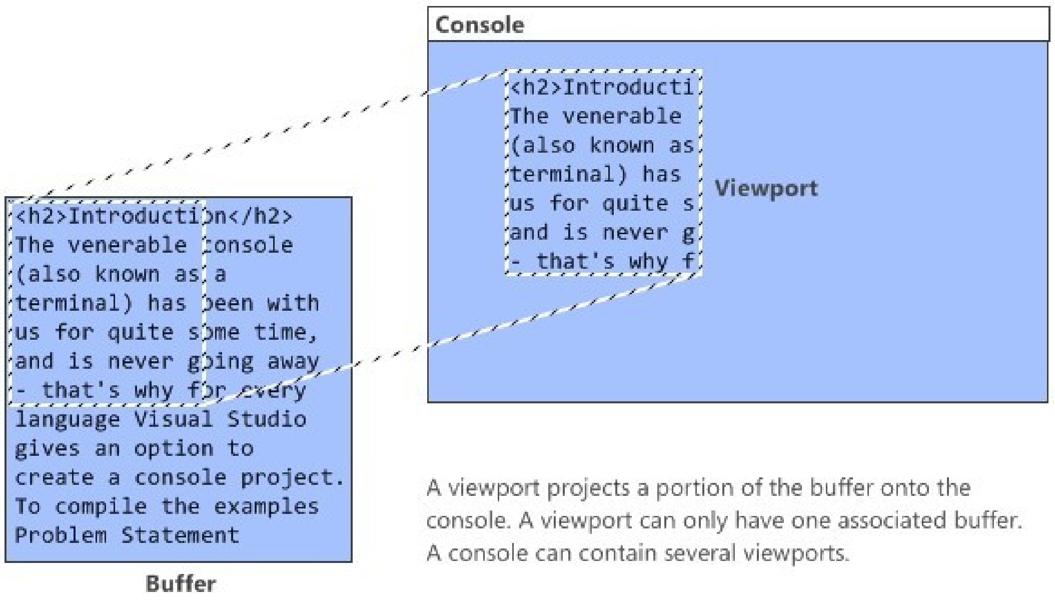
Typically, most users interact with a single buffer and viewport. It *is*, however, possible to have a console window where you have, say, the area split vertically between two viewports, each having their corresponding buffers. This can be done using utilities such as the `screen` Linux command.

An Advanced Terminal

One problem with a typical operating system terminal is that it is *extremely slow* if you pipe a lot of data into it. For example, a Windows terminal window (`cmd.exe`) uses GDI to render the characters, which is completely unnecessary. In a fast-paced trading environment, you want the rendering to be hardware-accelerated: characters should be presented as pre-rendered textures placed on a surface using an API such as OpenGL.[22]

A trading terminal consists of *multiple* buffers and viewports. In a typical setup, different buffers might be getting updated concurrently with data from various exchanges or trading bots, and all of this information needs to be presented on a single screen.

[22]We also use ASCII, since Unicode is rarely, if ever, required. Having 1 char = 1 byte is a good practice if you don't need to support extra character sets. While not relevant to the discussion at hand, it also greatly simplifies the implementation of string processing algorithms on both GPUs and FPGAs.


Console
Viewport
Buffer
A viewport projects a portion of the buffer onto the console. A viewport can only have one associated buffer. A console can contain several viewports.

```
<h2>Introduction</h2>
The venerable console
(also known as a
terminal) has been with
us for quite some time,
and is never going away
- that's why for every
language Visual Studio
gives an option to
create a console project.
To compile the examples
Problem Statement
```

Buffers also provide functionality that is a lot more exciting that just a 1D or 2D linear storage. For example, a `TableBuffer` might be defined as:

```
public class TableBuffer : IBuffer
{
  private readonly TableColumnSpec[] spec;
  private readonly int totalHeight;
  private readonly List<string[]> buffer;
  private static readonly Point invalidPoint = new Point(-1,-1);
  private readonly short[,] formatBuffer;

  public TableBuffer(TableColumnSpec [] spec, int totalHeight)
  {
    this.spec = spec;
    this.totalHeight = totalHeight;

    buffer = new List<string[]>();
    for (int i = 0; i < (totalHeight - 1); ++i)
    {
      buffer.Add(new string[spec.Length]);
    }
    formatBuffer = new short[spec.Max(s => s.Width),totalHeight];
  }

  public struct TableColumnSpec
  {
    public string Header;
    public int Width;
    public TableColumnAlignment Alignment;
  }
}
```

In other words, a buffer can take some specification and build a table (yes, a food old-fashioned ASCII-formatted table!) and present it on screen.[23]

A viewport is in charge of getting data from the buffer. Some of its characteristics include:

- A reference to the buffer it's showing.
- Its size.
- If the viewport is smaller than the buffer, it needs to specify which part of the buffer it is going to show. This is expressed in absolute x-y coordinates.
- The location of the viewport on the overall console window.
- The location of the cursor, assuming this viewport is currently taking user input.

Where's the Façade?

The console itself *is* the façade in this particular system. Internally, the console has to manage a lot of different internal settings:

[23]Many trading terminals have abandoned pure ASCII representation in favor of more mixed-mode approaches, such as simply using monospace fonts in ordinary UI controls or rendering many little text-based consoles in separate windowing API rather than sticking to a single canvas.

```
public class Console : Form
{
  private readonly Device device;
  private readonly PresentParameters pp;
  private IList<Viewport> viewports;
  private Size charSize;
  private Size gridSize;
  // many more fields here
}
```

Initialization of the console is also, typically, a very nasty affair. At the very least, you need to specify the size of individual characters and the width and height of the console (in terms of the number of characters). In some situations, you do actually want to specify console parameters in excruciating detail but, when in a hurry, you just want a sensible set of defaults.

However, since it's a Façade, it actually tries to give a really accessible API. This might either take a number of sensible parameters to initialize all the guts from.

```
private Console(bool fullScreen, int charWidth, int charHeight,
  int width, int height, Size? clientSize)
{
  int windowWidth =
    clientSize == null ? charWidth*width : clientSize.Value.Width;
  int windowHeight =
    clientSize == null ? charHeight*height : clientSize.Value.Height;

  // and a lot more code

  // single buffer and viewport created here
  // linked together and added to appropriate collections
  // image textures generated
  // grid size calculated depending on whether we want fullscreen mode
}
```

Alternatively, one might pack all those arguments into a single object which, again, has some sensible defaults:

```
public static Console Create(ConsoleCreationParameters ccp) { ... }

public class ConsoleCreationParameters
{
  public Size? ClientSize;
  public int CharacterWidth = 10;
  public int CharacterHeight = 14;
  public int Width = 20;
  public int Height = 30;
  public bool FullScreen;
  public bool CreateDefaultViewAndBuffer = true;
}
```

As you can see, with the façade we've built, there are no less than three ways of setting up the console:

- Use the low-level API to configure the console explicitly, viewports and buffers included.
- Use the `Console` constructor which requires you to provide fewer values, and makes a couple of useful assumptions (e.g., that you want just one viewport with an underlying buffer).
- Use the constructor that takes a `ConsoleCreationParameters` object. This requires you to provide even fewer pieces of information, as every field of that structure has a suitable default.

Summary

The Façade design pattern is a way of putting a simple interface in front of one or more complicated subsystems. As we have seen in the Magic Square example, in addition to providing a convenient interface, it's possible to expose the internal mechanics and allow for further customization by advanced users. Similarly, in our final example, a complicated set-up involving many buffers and viewports can be used directly or, if you just want a simple console with a single buffer and associated viewport, you can get it through a very accessible and intuitive API.

Flyweight

A Flyweight (also sometimes called a *token* or a *cookie*) is a temporary component which acts as a 'smart reference' to something. Typically, flyweights are used in situations where you have a very large number of very similar objects, and you want to minimize the amount of memory that is dedicated to storing all these values.

Let's take a look at some scenarios where this pattern becomes relevant.

User Names

Imagine a massively multiplayer online game. I bet you $20 there's more than one user called John Smith – quite simply because it is a popular name. So if we were to store that name over and over (in UTF-16), we would be spending 10 characters (plus a few more bytes for every `string`). Instead, we could store the name once and then store a reference to every user with that name. That's quite a saving.

Furthermore, the last name Smith is likely very popular on its own. Thus, instead of storing full names, splitting the name into first and last would allow further optimizations, as you could simply store `"Smith"` in an indexed store and then simply store the index value instead of the actual string.

Let's see how we can implement such a space-saving system. We're actually going to do this scientifically by forcing garbage collection and measuring the amount of memory taken using dotMemory.

So here's the first, naive implementation of a `User` class. Notice that the full name is kept as a single string.

```
public class User
{
  public string FullName { get; }

  public User(string fullName)
  {
    FullName = fullName;
  }
```

The implication of the above is that “John Smith” and “Jane Smith” are distinct strings, each occupying their own memory. Now we can construct an alternative type, `User2`, that is a little bit smarter in terms of its storage while exposing the same API (I’ve avoided extracting an `IUser` interface here for brevity):

```
public class User2
{
  private static List<string> strings = new List<string>();
  private int[] names;

  public User2(string fullName)
  {
    int getOrAdd(string s)
    {
      int idx = strings.IndexOf(s);
      if (idx != -1) return idx;
      else
      {
        strings.Add(s);
        return strings.Count - 1;
      }
    }

    names = fullName.Split(' ').Select(getOrAdd).ToArray();
  }

  public string FullName => string.Join(" ", names.Select(i => strings[i]));
}
```

As you can see, the actual strings are stored in a single `List`. The full name, as it is fed into the constructor, is split into the constituent parts. Each part gets inserted (unless it's already there) into the list of strings, and the `names` array simply stores the indices of the names in the list, however many there are. This means that, strings notwithstanding, the amount of non-static memory `User2` takes up is 64 bits (two `Int32`s).

Now is a good time to pause and explain where exactly the Flyweight is. Essentially, the flyweight is the index that we are storing. A flyweight is a tiny object with a very small memory footprint that points to something larger that is stored elsewhere.

The only question remaining is whether or not this approach actually makes sense. While it's very difficult to simulate this on actual users (this would require a live data set), we are going to do the following:

- Generate 100 first and 100 last names as random strings. The algorithm for making a random string is as follows:

```
public static string RandomString()
{
  Random rand = new Random();
  return new string(
    Enumerable.Range(0, 10)
              .Select(i => (char) ('a' + rand.Next(26))).ToArray());
}
```

- Next up, we make a concatenation (cross-product) of every first and last name and initialize 100x100 users:

```
var users = new List<User>(); // or User2
foreach (var firstName in firstNames)
foreach (var lastName in lastNames)
  users.Add(new User($"{firstName} {lastName}"));
```

- Just to be safe, we force GC at this point.

- Finally, we use the dotMemory unit testing API to output the total amount of memory taken up by the program.

Running this entirely unscientific (but indicative) test on my machine tells me that the `User2` implementations saves us 329,305 bytes. Is this significant? Well, let's try to calculate: a single 10-character string takes up 34 bytes (14 bytes[24] + 2x10 bytes for the letters), so 340,000 bytes for all the strings. This means we reduced the amount of memory taken by 97%! If this isn't cause for celebration, I don't know what is.

Text Formatting

Say you're working with a text editor, and you want to add formatting to text – for example, make text bold, italic or capitalize it. How would you do this? One option is to treat each character individually: if your text is composed of X characters, you make a `bool` array of size X and simply flip each of the flags if you want to alter text. This would lead to the following implementation:

```
public class FormattedText
{
  private string plainText;

  public FormattedText(string plainText)
  {
    this.plainText = plainText;
    capitalize = new bool[plainText.Length];
  }

  public void Capitalize(int start, int end)
  {
    for (int i = start; i <= end; ++i)
      capitalize[i] = true;
  }
```

[24]The size of a `string` actually depends on the bitness of the operating system as well as the version of .NET that you are using.

```
  private bool[] capitalize;
}
```

I'm using capitalization here (because that's what a text console can render), but you can think of other forms of formatting being here too. For every type of formatting, you'd be making another Boolean array, initializing it to the right size in the constructor (and imagine the nightmare if the text changes!) and then, of course, you would need to take into account those Boolean flags whenever you actually want to show the text somewhere:

```
public override string ToString()
{
  var sb = new StringBuilder();
  for (var i = 0; i < plainText.Length; i++)
  {
    var c = plainText[i];
    sb.Append(capitalize[i] ? char.ToUpper(c) : c);
  }
  return sb.ToString();
}
```

This approach does in fact work:

```
var ft = new FormattedText("This is a brave new world");
ft.Capitalize(10, 15);
WriteLine(ft); // This is a BRAVE new world
```

But of course we are wasting memory. Even if the text has *no* formatting whatsoever, we still allocated the array. True, we could have made it lazy so that it's only created whenever someone uses the `Capitalize()` method, but then we would still lose a lot of memory on first use, particularly with large texts.

This is precisely the situation the Flyweight design pattern is made for! In this particular case, we're going to define a flyweight as a `Range` class that stores information about the start and end position of a substring within a string, as well as all the formatting information we desire:

```
public class TextRange
{
  public int Start, End;
  public bool Capitalize; // also Bold, Italic, etc.

  public bool Covers(int position)
  {
    return position >= Start && position <= End;
  }
}
```

Now, we can define a `BetterFormattedText` class that simply stores a list of all the formatting that was applied:

```
public class BetterFormattedText
{
  private readonly string plainText;
  private readonly List<TextRange> formatting
    = new List<TextRange>();

  public BetterFormattedText(string plainText)
  {
    this.plainText = plainText;
  }

  public TextRange GetRange(int start, int end)
  {
    var range = new TextRange {Start = start, End = end};
    formatting.Add(range);
    return range;
  }

  public class TextRange { ... }
}
```

Notice that `TextRange` is an inner class – this is a design decision, and you could easily keep it external. Now, instead of a dedicated `Capitalize()` method, we simply have a method called `GetRange()` that does three things: it creates a new range, adds it to a list of formatting, but also returns it to the client to be operated upon.

All that remains now is to make a new implementation of `ToString()` that incorporates this flyweight-based approach. Here it is:

```
public override string ToString()
{
  var sb = new StringBuilder();

  for (var i = 0; i < plainText.Length; i++)
  {
    var c = plainText[i];
    foreach (var range in formatting)
      if (range.Covers(i) && range.Capitalize)
        c = char.ToUpperInvariant(c);
    sb.Append(c);
  }

  return sb.ToString();
}
```

As you can see, we simply iterate each of the characters. For each character, we check all the ranges with the `Covers()` method, and if that range covers this point and has special formatting, we show that formatting to the end user. Here is how you would use the new API:

```
var bft = new BetterFormattedText("This is a brave new world");
bft.GetRange(10, 15).Capitalize = true;
WriteLine(bft); // This is a BRAVE new world
```

Admittedly, ours is a fairly inefficient implementation of Flyweight (traversal of every character is just too tedious), but hopefully it’s obvious that the general approach saves a lot of memory in the long run.

Summary

The Flyweight pattern is fundamentally a space-saving technique. Its exact incarnations are diverse: sometimes you have the Flyweight being returned as an API token that allows you to perform modifications of whoever has spawned it, whereas at other times the Flyweight is implicit, hiding behind the scenes – as in the case of our `User`, where the client isn't meant to know about the Flyweight actually being used.

In the .NET Framework, the principal Flyweight-like object is, of course, `Span<T>`. Just like the `TextRange` we implemented when working with strings, `Span<T>` is a type that has information about a part of an array: the starting position and length. Operations on the `Span` get applied to the object the `Span` refers to, and .NET provides a rich API for creating spans on different types of objects. `Span` also makes heavy use of C# 7's `ref`-related APIs (such a `ref` returns).

Proxy

When we looked at the Decorator design pattern, we saw the different ways of enhancing the functionality of an object. The Proxy design pattern is similar, but its goal is generally to preserve exactly (or as closely as possible) the API that is being used while offering certain internal enhancements.

Proxy is an unusual design pattern in that it isn't really homogeneous. The many different kinds of proxies people build are quite numerous and serve entirely different purposes. In this chapter we'll take a look at a selection of different proxy objects, and you can find more online.

Protection Proxy

The idea of a protection proxy, as the name suggests, is to provide access control to an existing object. For example, you might be starting out with an object called `Car` that has a single `Drive()` method that lets you drive the car (here we go, another synthetic example).

```
public class Car // : ICar
{
  public void Drive()
  {
    WriteLine("Car being driven");
  }
}
```

But, later on, you decide that you want to only let people drive the car if they are old enough. What if you don't want to change `Car` itself and you want the extra checks to be done somewhere else (SRP)? Let's see… first, you extract the "ICar `interface (note this operation doesn't affect Car`‘ in any significant way):

```
public interface ICar
{
  void Drive();
}
```

The protection proxy we're going to build is going to depend on a `Driver` that's defined like this:

```
public class Driver
{
  public int Age { get; set; }

  public Driver(int age)
  {
    Age = age;
  }
}
```

The proxy itself is going to take a `Driver` in the constructor and it's going to expose the same `ICar` interface as the original car, the only difference being that some internal checks occur making sure the driver is old enough:

```
public class CarProxy : ICar
{
  private Car car = new Car();
  private Driver driver;

  public CarProxy(Driver driver)
  {
    this.driver = driver;
  }

  public void Drive()
  {
    if (driver.Age >= 16)
      car.Drive();
    else
    {
      WriteLine("Driver too young");
    }
  }
}
```

Here is how one would use this proxy:

```
ICar car = new CarProxy(new Driver(12));
car.Drive(); // Driver too young
```

There's one piece of the puzzle that we haven't really addressed. Even though both `Car` and `CarProxy` implement `ICar`, their constructors are not identical! This means that, strictly speaking, the interfaces of the two objects are not strictly identical. Is this a problem? This depends:

- If your code was dependent on `Car` rather than `ICar` (violating DIP) then you would need to search-and-replace every use of this type in your code. Not impossible with tools like ReSharper/Rider, just really annoying.
- If your code was dependent on `ICar` but you were explicitly invoking `Car` constructors, you would have to find all those constructor invocations and feed each of them a `Driver`.
- If you were using dependency injection, you are good to go provided you register a `Driver` in the container.

So, among other things, the protection proxy we've built is an illustration of the benefits of using an IoC container with constructor injection support.

Property Proxy

C# makes the use of properties easy: you can use either 'full' or automatic properties, and now there's expression-based notation for getters and setters, so you can keep properties really concise. However, that's not always what you want: sometimes, you want a getter or setter of each property in your code to do something in addition to just the default actions. For example, you might want setters that prevent self-assignment and also (for illustrative purposes) output some info about what value is being assigned and to what property.

So instead of using ordinary properties, you might want to introduce a *property proxy* – a class that, for all intents and purposes, behaves like a property but is actually a separate class with domain-specific behaviors (and associated performance costs). You would start building this class by wrapping a simple value and adding whatever extra information you want the property to have (e.g., the property name):

```
public class Property<T> where T : new()
{
  private T value;
  private readonly string name;

  public T Value
  {
    get => value;

    set
    {
      if (Equals(this.value, value)) return;
      Console.WriteLine($"Assigning {value} to {name}");
      this.value = value;
    }
  }

  public Property() : this(default(T)) {}

  public Property(T value, string name = "")
  {
    this.value = value;
    this.name = name;
  }
}
```

For now, all we have is a simple wrapper, but where's the *proxy* part of it all? After all, we want a `Property<int>` to behave as close to an `int` as possible. To that end, we can define a couple of implicit conversion operators:

```
public static implicit operator T(Property<T> property)
{
  return property.Value; // int n = p_int;
}

public static implicit operator Property<T>(T value)
{
  return new Property<T>(value); // Property<int> p = 123;
}
```

The first operator lets us implicitly convert the property type to its underlying value, the second operator lets us initialize a property from a value (without a `name`, of course). Sadly, C# does not allow us to override the assignment = operator.

How would you use this property proxy? Well, there are two ways I can think of. One, and the most obvious, is to expose the property as a public field.

```
public class Creature
{
  public Property<int> Agility
    = new Property<int>(10, nameof(Agility))
}
```

Unfortunately, this approach is not a 'proper' proxy because, while it replicates the interface of an ordinary property, it doesn't give us the behavior we want:

```
var c = new Creature();
c.Agility = 12; // <nothing happens!>
```

When you assign a value, as you would with an ordinary property, absolutely nothing happens. Why? Well, the reason is that we invoked the implicit conversion operator which, instead of changing an existing property, just gave us a new property instead! It's definitely not what we wanted and, furthermore, we've lost the `name` value as it was never propagated by the operator.

So the solution here, if we really want the property to both look like a duck and quack like a duck, is to create a wrapper (delegating) property, and keep the proxy as a private backing field:

```
public class Creature
{
  public readonly Property<int> agility
    = new Property<int>(10, nameof(agility));

  public int Agility
  {
    get => agility.Value;
    set => agility.Value = value;
  }
}
```

With this approach, we finally get the desired behavior:

```
var c = new Creature();
c.Agility = 12; // Assigning 12 to Agility
```

Purists might argue that this isn't an ideal proxy (since we've had to generate both a new class as well as rewrite an existing property), but this is purely a limitation of the C# programming language.

Value Proxy

A value proxy is a proxy around a primitive value, such as an integer. Why would you want such a proxy? Well, it's because certain primitive values can have special meanings.

Consider percentages. Multiplying by 50 is different from multiplying by 50% because the latter is really multiplication by 0.5. But you still want to refer to 50% as 50% in your code, right? Let's see if we can build a `Percentage` type.

First of all, we need to agree on construction. Let's assume that we do, in fact, store a `decimal` behind the scenes that is actually a multiplier. In that case, we can begin our `Percentage` class as follows:

```
[DebuggerDisplay("{value*100.0f}%")]
public struct Percentage
{
  private readonly decimal value;

  internal Percentage(decimal value)
  {
    this.value = value;
  }
  // more members here
}
```

We have different choices for how to actually construct percentage values. One approach would be to adopt extension methods:

```
public static class PercentageExtensions
{
  public static Percentage Percent(this int value)
  {
    return new Percentage(value/100.0m);
  }

  public static Percentage Percent(this decimal value)
  {
    return new Percentage(value/100.0m);
  }
}
```

We want this percentage to act the same as percentage values in, e.g., Microsoft Excel. Multiplying by 50% should effectively multiply by 0.5; other operations should work in a similar fashion. Thus, we need to define many operators such as:

```
public static decimal operator *(decimal f, Percentage p)
{
  return f * p.value;
}
```

Let's not forget that percentage can also operate on other percentages: for example, you can add 5% and 10% together and, similarly, you can take 50% of 50% (getting 25%). So you need more operators such as:

```
public static Percentage operator +(Percentage a, Percentage b)
{
  return new Percentage(a.value + b.value);
}
```

In addition, you need the usual trappings: `Equals()`, `GetHashCode()`, a meaningful `ToString()` such as:

```
public override string ToString()
{
  return $"{value*100}%";
}
```

And that's your value proxy. Now, if you need to operate on percentages in your application and stored them explicitly as percentages, you can do so.

```
Console.WriteLine(10m * 5.Percent());          // 0.50
Console.WriteLine(2.Percent() + 3m.Percent()); // 5.00%
```

Composite Proxy: SoA/AoS

Many applications, such as game engines, are very sensitive to data locality. For example, consider the following class:

```
class Creature
{
  public byte Age;
  public int X, Y;
}
```

If you had several creates in your game, kept in an array, the memory layout of your data would appear as

```
Age X Y Age X Y Age X Y ... and so on
```

This means that, if you wanted to update the X co-ordinate of all objects in an array, your iteration code would have to jump over the other fields to get each of the Xs.

Turns out that CPUs generally like data locality, i.e., data being kept together. This is often called the AoS/SoA (Array of Structures/Structure of Arrays) problem. For us, it would be much better if the memory layout was in SoA form, as follows:

```
Age Age Age ... X X X ... Y Y Y
```

How can we achieve this? Well, we can build a data structure that keep exactly such a layout and then expose `Creature` objects as proxies.

Here's what I mean. First of all, we create a `Creatures` collection (I'll used arrays as underlying data types) that enforces data locality for each of the 'fields':

```
class Creatures
{
  private readonly int size;
  private byte [] age;
  private int[] x, y;

  public Creatures(int size)
  {
    this.size = size;
    age = new byte[size];
    x = new int[size];
    y = new int[size];
  }
}
```

Now, a `Creature` type can be constructed as a *hollow proxy* (a stateless proxy/memento amalgamation) that points to an element within the `Creatures` container.

```
public struct Creature
{
  private readonly Creatures creatures;
  private readonly int index;

  public Creature(Creatures creatures, int index)
  {
    this.creatures = creatures;
    this.index = index;
  }

  public ref byte Age => ref creatures.age[index];
  public ref int X => ref creatures.x[index];
  public ref int Y => ref creatures.y[index];
}
```

Note that the above is class is *nested* within `Creatures`. The reason for this is that its property getters need to access `private` members of `Creatures`, which would be impossible if the class was on the same scope as the container.

So now that we have this proxy, we can give the `Creatures` container additional features, such as an indexer or a `GetEnumerator()` implementation:

```
public class Creatures
{
  // members here
  public Creature this[int index]
    => new Creature(this, index);

  public IEnumerator<Creature> GetEnumerator()
  {
    for (int pos = 0; pos < size; ++pos)
      yield return new Creature(this, pos);
  }
}
```

And that's it! We can now look at a side-by-side comparison of the AoS approach and the new SoA approach:

```
// AoS
var creatures = new Creature[100];
foreach (var c in creatures)
{
  c.X++; // not memory-efficient
}

// SoA
var creatures2 = new Creatures(100);
foreach (var c in creatures2)
{
  c.X++;
}
```

Naturally, what I've shown here is a simplistic model. It would be more useful with arrays replaced by more flexible data structures like `List<T>`, and a lot more features could be added in order to make `Creatures` even more user-friendly.

Composite Proxy with Array-Backed Properties

Suppose you're working on an application that generates bricklaying designs. You need to decide what surfaces you want to cover with bricks, so you make a list of check boxes as follows:

☒ Pillars

☒ Walls

☒ Floors

☒ All

Most of these are easy and can be bound 1-to-1 to `boolean` variables, but the last option, `All`, cannot. How would you implement it in code? Well, you could try something like the following:

```
public class MasonrySettings
{
  public bool Pillars, Walls, Floors;

  public bool All
  {
    get { return Pillars && Walls && Floors; }
    set {
      Pillars = value;
      Walls = value;
      Floors = value;
    }
  }
}
```

This implementation might work, but it's not 100% correct. The last checkbox called `All` is actually not even `boolean` because it can be in 3 states:

- Checked if all items are checked
- Unchecked if all items are unchecked
- Grayed if some items are checked and others are not

This makes it a bit of a challenge: how do we make a variable for the state of this element that can be reliably bound to UI?

First of all, those combinations using && are ugly. We already have a tool called Array-Backed Properties that can help us take care of that, transforming the class to:

```
public class MasonrySettings
{
  private bool[] flags = new bool[3];

  public bool Pillars
  {
    get => flags[0];
    set => flags[0] = value;
  }

  // similar for Floors and Walls
}
```

Now, care to guess that type the `All` variable should have? Personally I'd go for a `bool?` (a.k.a. `Nullable<bool>`) where the `null` can indicate an indeterminate state. This means that we check the homogeneity of each element in the array and return its first element if it's homogeneous (i.e., all elements are the same) and `null` otherwise:

```
public bool? All
{
  get
  {
    if (flags.Skip(1).All(f => f == flags[0]))
      return flags[0];
    return null;
  }

  set
  {
    if (!value.HasValue) return;
    for (int i = 0; i < flags.Length; ++i)
      flags[i] = value.Value;
  }
}
```

The getter above is fairly self-explanatory. When it comes to the setter, its value gets assigned to every element in the array. If the passed in value is `null`, we don't do anything. An alternative implementation could, for example, flip every boolean member inside the array – your choice!

Virtual Proxy

There are situations where you only want the object constructed when it's accessed, and you don't want to allocate it prematurely. If this was your starting strategy, you would typically use a `Lazy<T>` or similar mechanism, feeding the initialization code into its constructor lambda. However, there are situations when you are adding lazy instantiation at a later point in time, and you cannot change the existing API.

In this situation, what you end up building is a *virtual proxy*: an object that has the same API as the original, giving the appearance of an instantiated object, but behind the scenes the proxy only instantiates the object when it's actually necessary.

Imagine a typical image interface:

```
interface IImage
{
  void Draw();
}
```

An eager (opposite of lazy) implementation of a `Bitmap` (nothing to do with `System.Drawing.Bitmap`!) would load the image from a file on construction, even if that image isn't actually required for anything. And yes, the code below is an emulation.

```
class Bitmap : IImage
{
  private readonly string filename;

  public Bitmap(string filename)
  {
    this.filename = filename;
    WriteLine($"Loading image from {filename}");
  }

  public void Draw()
  {
    WriteLine($"Drawing image {filename}");
  }
}
```

The very act of construction of this `Bitmap` will trigger the loading of the image:

```
var img = new Bitmap("pokemon.png");
// Loading image from pokemon.png
```

That's not quite what we want. What we want is the kind of bitmap that only loads itself when the `Draw()` method is used. Now, I suppose we could jump back into `Bitmap` and make it lazy, but we're going to assume the original implementation is set in stone and is not modifiable.

So what we can then build is a virtual proxy that will use the original `Bitmap`, provide an identical interface, and also reuse the original `Bitmap`'s functionality:

```
class LazyBitmap : IImage
{
  private readonly string filename;
  private Bitmap bitmap;

  public LazyBitmap(string filename)
  {
    this.filename = filename;
  }
  public void Draw()
  {
    if (bitmap == null)
      bitmap = new Bitmap(filename);

    bitmap.Draw();
  }
}
```

Here we are. As you can see, the constructor of this `LazyBitmap` is a lot less 'heavy': all it does is store the name of the file to load the image from, and that's it – the image doesn't actually get loaded.[25]

All of the magic happens in `Draw()`: this is where we check the `bitmap` reference to see whether the underlying (eager!) bitmap has been constructed. If it hasn't, we construct it, and then call its `Draw()` function to actually draw the image.

Now imagine you have some API that uses an `IImage` type:

```
public static void DrawImage(IImage img)
{
  WriteLine("About to draw the image");
  img.Draw();
  WriteLine("Done drawing the image");
}
```

We can use that API with an instance of `LazyBitmap` instead of `Bitmap` (hooray, polymorphism!) to render the image, loading it in a lazy fashion:

[25]Not that it matters for this particular example, but this implementation is not thread-safe. Imagine two threads that both do a `null` check, pass it, and then both assign `bitmap` one after another – the constructor will be called twice. This is why we use `System.Lazy`, which is thread-safe by design.

```
var img = new LazyBitmap("pokemon.png");
DrawImage(img); // image loaded here

// About to draw the image
// Loading image from pokemon.png
// Drawing image pokemon.png
// Done drawing the image
```

Communication Proxy

Suppose you call a method `Foo()` on an object of type `Bar`. Your typical assumption is that `Bar` has been allocated on the same machine as the one running your code, and you similarly expect `Bar.Foo()` to execute in the same process.

Now imagine that you make a design decision to move `Bar` and all its members off to a different machine on the network. But you still want the old code to work! If you want to keep going as before, you'll need a *communication proxy* – a component that proxies the calls 'over the wire' and of course collects results, if necessary.

Let's implement a simple ping-pong service to illustrate this. First, we define an interface:

```
interface IPingable
{
  string Ping(string message);
}
```

If we are building ping-pong in-process, we can implement Pong as follows:

```
class Pong : IPingable
{
  public string Ping(string message)
  {
    return message + " pong";
  }
}
```

Basically, you ping a Pong and it appends the word " pong" to the end of the message and returns that message. Notice that I'm not using a StringBuilder here, but instead making a new string on each turn: this lack of mutation helps replicate this API as a web service.

We can now try out this set-up and see how it works in-process:

```
void UseIt(IPingable pp)
{
  WriteLine(pp.ping("ping"));
}

Pong pp = new Pong();
for (int i = 0; i < 3; ++i)
{
  UseIt(pp);
}
```

And the end result is that we print "ping pong" three times, just as we wanted.

So now, suppose you decide to relocate the Pingable service to a web server, far-far away. Perhaps you even decide to expose it through a special framework such as ASP.NET:

```
[Route("api/[controller]")]
public class PingPongController : Controller
{
  [HttpGet("{msg}")]
  public string Get(string msg)
  {
    return msg + " pong";
  }
}
```

With this set-up, we'll build a communication proxy called `RemotePong` that will be used in place of `Pong`.

```
class RemotePong : IPingable
{
  string Ping(string message)
  {
    string uri = "http://localhost:9149/api/pingpong/" + message;
    return new WebClient().DownloadString(uri);
  }
}
```

With this implemented, we can now make a single change:

```
RemotePong pp; // was Pong
for (int i = 0; i < 3; ++i)
{
  UseIt(pp);
}
```

And that's it, you get the same output, but the actual implementation can be running on Kestrel in a Docker container somewhere halfway around the world.

Dynamic Proxy for Logging

Say you're testing a piece of code and you want to record the number of times particular methods are called, and what arguments they are called with. You have a couple of options, including:

- Using AOP approaches such as PostSharp or Fody to create assemblies where the required functionality is weaved into the code.
- Using profiling/tracing software instead.
- Creating dynamic proxies for your objects in tests.

A *dynamic proxy* is a proxy created at runtime. It allows us to take an existing object and, provided a few rules are followed, to override or wrap some of its behaviors to perform additional operations.

So imagine that you are writing tests that cover the operation of a `BankAccount`, a class that implements the following interface:

```
public interface IBankAccount
{
  void Deposit(int amount);
  bool Withdraw(int amount);
}
```

Suppose that your starting point is a test such as the following:

```
var ba = new BankAccount();
ba.Deposit(100);
ba.Withdraw(50);
WriteLine(ba);
```

When performing those operations, you also want a *count* of the number of methods that have been called. So, effectively, you want to wrap a `BankAccount` with some sort of dynamically constructed proxy that implements the `IBankAccount` interface and keeps a log of all the methods called.

We shall construct a new class that we'll call `Log<T>` that is going to be a dynamic proxy for any type `T`:

```
public class Log<T> : DynamicObject
  where T : class, new()
{
  private readonly T subject;
  private Dictionary<string, int> methodCallCount =
    new Dictionary<string, int>();

  protected Log(T subject)
  {
    this.subject = subject;
  }
}
```

Our class takes a `subject`, which is the class it's wrapping, and has a simple dictionary of method call counts.

Now, the class above inherits from `DynamicObject`, which is great because we want to make a log of the calls made to its various methods, and only then actually invoke those methods. Here's how we can implement this:

```
public override bool TryInvokeMember(InvokeMemberBinder binder, object[] args, \
out object result)
{
  try
  {
    if (methodCallCount.ContainsKey(binder.Name))
      methodCallCount[binder.Name]++;
    else
      methodCallCount.Add(binder.Name, 1);

    result = subject
      ?.GetType()
      ?.GetMethod(binder.Name)
      ?.Invoke(subject, args);
    return true;
  }
  catch
  {
    result = null;
    return false;
  }
}
```

As you can see, all we're doing is logging the number of calls made to a particular method, then invoking the method itself using reflection.

Now, there's only one tiny problem for us to handle: how do we get our `Log<T>` to pretend as if it were implementing some interface `I`? This is where dynamic proxy frameworks come in. The one we're going to be using is called ImpromptuInterface[26]. This framework has a method called `ActLike()` that allows a `dynamic` object to pretend that it is of a particular interface type.

Armed with this, we can give our `Log<T>` a static factory method that would construct a new instance of `T`, wrap it in a `Log<T>` and then expose it as some interface `I`:

```
public static I As<I>() where I : class
{
  if (!typeof(I).IsInterface)
    throw new ArgumentException("I must be an interface type");

  // duck typing here!
  return new Log<T>(new T()).ActLike<I>();
}
```

The end result of it all is that we can now perform a simple replacement and get a record of all the calls made to a the bank account class:

[26]You can get it from NuGet; the source code is at https://github.com/ekonbenefits/impromptu-interface.

```
//var ba = new BankAccount();
var ba = Log<BankAccount>.As<IBankAccount>();

ba.Deposit(100);
ba.Withdraw(50);

WriteLine(ba);
// Deposit called 1 time(s)
// Withdraw called 1 time(s)
```

Naturally, in order for the above to work, I've overridden `Log<T>.ToString()` to output call counts. Sadly, the wrapper we've made will not automatically proxy over calls to `ToString()`/`Equals()`/`GetHashCode()` because every `object` has them intrinsically built in. If you do want to connect these to the underlying, you'd have to add overrides in `Log<T>` and then use the `subject` field to make the appropriate calls.

Summary

This chapter has presented a number of proxies. Unlike the Decorator pattern, the Proxy doesn't try to expand the public API surface of an object by adding new members (unless it can't be helped). All it tries to do is enhance the underlying behavior of existing members.

Plenty of different proxies exist:

- Property proxies are stand-in objects that can replace fields and perform additional operations during assignment and/or access.
- Virtual proxies provide virtual access to the underlying object, and can implement behaviors such as lazy object loading. You may feel like you're working with a real object, but the underlying implementation may not have been created yet, and can, for example, be loaded on demand.
- Communication proxies allow us to change the physical location of the object (e.g., move it to the cloud) but allow us to use pretty much the same API. Of course, in this case the API is just a shim for a remote service such as some available REST API.

- Logging proxies allow you to perform logging in addition to calling the underlying functions.

There are lots of other proxies out there, and chances are that the ones you build yourself will not fall into a preexisting category, but will instead perform some action specific to your domain.

Behavioral Patterns

When most people hear about behavioral patterns, it's mainly with relation to animals and how to get them to do what you want. Well, in a way, all of coding is about programs doing what you want, so behavioral software design patterns cover a very wide range of behaviors that are, nonetheless, quite common in programming.

As an example, consider the domain of software engineering. We have languages which are compiled, which involves lexing, parsing and a million other things (the Interpreter pattern) and, having constructed an abstract syntax tree (AST) for a program, you might want to analyze the program for possible bugs (the Visitor pattern). All of these are behaviors that are common enough to be expressed as patterns, and this is why we are here today.

Unlike Creational patterns (which are concerned exclusively with the creation of objects) or Structural patterns (which are concerned with composition/aggregation/inheritance of objects), Behavioral design patterns do not follow a central theme. While there are certain similarities between different patterns (e.g., Strategy and Template Method do the same thing in different ways), most patterns present unique approaches to solving a particular problem.

Chain of Responsibility

Consider the typical example of corporate malpractice: insider trading. Say a particular trader has been caught red-handed trading on inside information. Who is to blame for this? If management didn't know, it's the trader. But maybe the trader's peers were in on it, in which case the group manager might be the one responsible. Or perhaps the practice is institutional, in which case it's the CEO who would take the blame.[27]

The above is an example of a responsibility chain: you have several different elements of a system who can all process a message one after another. As a concept, it's rather easy to implement, since all that's implied is the use of a list.

Scenario

Imagine a computer game where each creature has a name and two characteristic values – `Attack` and `Defense`:

```
public class Creature
{
  public string Name;
  public int Attack, Defense;

  public Creature(string name, int attack, int defense) { ... }
}
```

Now, as the creature progresses through the game, it might pick up an item (e.g., a magic sword), or it might end up getting enchanted. In either case, its attack and defense values will be modified by something we'll call a `CreatureModifier`.

[27] In all likelihood, if we're talking about banking, *nobody* gets punished. Nobody was punished for the subprime mortgage crisis. In the LIBOR fixing scandal only one trader got convicted (six bankers were accused in the UK but later cleared). What does this have to do with design patterns? Absolutely nothing! Just wanted to share.

Furthermore, situations where *several* modifiers are applied are not uncommon, so we need to be able to stack modifiers on top of a creature, allowing them to be applied in the order they were attached.

Let's see how we can implement this.

Method Chain

In the classic Chain of Responsibility implementation, we shall define `CreatureModifier` as follows:

```
public class CreatureModifier
{
  protected Creature creature;
  protected CreatureModifier next;

  public CreatureModifier(Creature creature)
  {
    this.creature = creature;
  }

  public void Add(CreatureModifier cm)
  {
    if (next != null) next.Add(cm);
    else next = cm;
  }

  public virtual void Handle() => next?.Handle();
}
```

There are a lot of things happening here, so let's discuss them in turn:

- The class takes and stores a reference to the `Creature` it plans to modify.
- The class doesn't really do much, but it's not abstract: all its members have implementations.
- The `next` member points to an optional `CreatureModifier` following this one. The implication is, of course, that the modifier can also be some inheritor of `CreatureModifier`.

- The `Add()` method adds another creature modifier to the modifier chain. This is done iteratively: if the current modifier is `null` we set it to that, otherwise we traverse the entire chain and put it on the end. Naturally this traversal has *O(n)* complexity.
- The `Handle()` method simply handles the next item in the chain, if it exists; it has no behavior of its own. The fact that it's `virtual` implies that it's meant to be overridden.

So far, all we have is an implementation of a poor man's append-only singly linked list. But when we start inheriting from it, things will hopefully become more clear. For example, here is how you would make a modifier that would double the creature's `attack` value:

```
public class DoubleAttackModifier : CreatureModifier
{
  public DoubleAttackModifier(Creature creature)
    : base(creature) {}

  public override void Handle()
  {
    WriteLine($"Doubling {creature.Name}'s attack");
    creature.Attack *= 2;
    base.Handle();
  }
}
```

All right, finally we're getting somewhere. So this modifier inherits from `CreatureModifier`, and in its `Handle()` method does two things: doubles the attack value and calls `Handle()` from the base class. That second part is critical: the only way in which a *chain* of modifiers can be applied is if every inheritor doesn't forget to call the base at the end of its own `Handle()` implementation.

Here is another, more complicated modifier. This modifier increases the defense of creatures with `attack` of 2 or less by 1:

```
public class IncreaseDefenseModifier : CreatureModifier
{
  public IncreaseDefenseModifier(Creature creature)
    : base(creature) {}

  public override void Handle()
  {
    if (creature.Attack <= 2)
    {
      WriteLine($"Increasing {creature.Name}'s defense");
      creature.Defense++;
    }

    base.Handle();
  }
}
```

Again we call the base class at the end. Putting it all together, we can now make a creature and apply a combination of modifiers to it:

```
var goblin = new Creature("Goblin", 1, 1);
WriteLine(goblin); // Name: Goblin, Attack: 1, Defense: 1

var root = new CreatureModifier(goblin);
root.Add(new DoubleAttackModifier(goblin));
root.Add(new DoubleAttackModifier(goblin));
root.Add(new IncreaseDefenseModifier(goblin));

// eventually...
root.Handle();
WriteLine(goblin); // Name: Goblin, Attack: 4, Defense: 1
```

As you can see, the above goblin is a 4/1 because its attack got doubled and the defense modifier, while added, did not affect its defense score.

Here's another curious point. Suppose you decide to cast a spell on a creature such that no bonus can be applied to it. Is it easy to do? Quite easy, actually, because all you have to do is avoid calling the base `handle()`: this avoids executing the entire chain:

```
public class NoBonusesModifier : CreatureModifier
{
  public NoBonusesModifier(Creature creature)
    : base(creature) {}

  public override void Handle()
  {
    WriteLine("No bonuses for you!");
    // no call to base.Handle() here
  }
}
```

That's it! Now, if you slot the `NoBonusesModifier` at the *beginning* of the chain, no further elements will be applied.

Broker Chain

The example with the pointer chain is very artificial. In the real world, you'd want creatures to be able to take on and lose bonuses arbitrarily, something which an append-only linked list doesn't support. Furthermore, you don't want to modify the underlying creature stats permanently (as we did) – instead, you want to keep modifications temporary.

One way to implement Chain of Responsibility is through a centralized component. This component can keep a list of *all* modifiers available in the game, and can facilitate queries for a particular creature's attack or defense by ensuring that all relevant bonuses are applied.

The component that we are going to build is called an *event broker*. Since it's connected to every participating component is represents the Mediator design pattern and, further, since it responds to queries through events, it leverages the Observer design pattern.

Let's build one. First of all, we'll define a structure called `Game` that will represent, well, a game that's being played:

```
public class Game // mediator pattern
{
  public event EventHandler<Query> Queries; // effectively a chain

  public void PerformQuery(object sender, Query q)
  {
    Queries?.Invoke(sender, q);
  }
}
```

The class Game is what we generally call an *event broker*: a central component that brokers (passes) events between different parts of the system. Here it is implemented using ordinary .NET events, but you can equally imagine an implementation using some sort of message queue.

In the game, we are using an event called called `Queries`. Essentially, this lets us raise this event and have it handled by every subscriber (listening component). But what do events have to do with querying a creature's attack or defense?

Well, imagine that you want to query a creature's statistic. You could certainly try to read a field, but remember – we need to apply all the modifiers before the final value is known. So instead we'll encapsulate a query in a separate object (this is the Command pattern[28]) defined as follows:

```
public class Query
{
  public string CreatureName;
  public enum Argument
  {
    Attack, Defense
  }
  public Argument WhatToQuery;
  public int Value; // bidirectional!
}
```

[28] Actually, there's a bit of confusion here. The concept of Command Query Separation (CQS) suggests the separation of operations into commands (which mutate state and yield no value) and queries (which do not mutate anything but yield a value). The GoF does not have a concept of a Query, so we let *any* encapsulated instruction to a component be called a Command.

All we've done in the above class is encapsulated the concept of querying a particular value from a creature. All we need to provide is the name of the creature and which statistic we're interested in. It is precisely this value (well, a reference to it) that will be constructed and used by `Game.Queries` to apply the modifiers and return the final `Value`.

Now, let's move on to the definition of `Creature`. It's very similar to what we had before. The only difference in terms of fields is a reference to a `Game`:

```
public class Creature
{
  private Game game;
  public string Name;
  private int attack, defense;

  public Creature(Game game, string name, int attack, int defense)
  {
    // obvious stuff here
  }
  // other members here
}
```

Now, notice how `attack` and `defense` are private fields now. This means that, to get at the *final* (post-modifier) attack value you would need to call a separate read-only property, for example:

```
public int Attack
{
  get
  {
    var q = new Query(Name, Query.Argument.Attack, attack);
    game.PerformQuery(this, q);
    return q.Value;
  }
}
```

This is where the magic happens! Instead of just returning a value or statically applying some reference-based chain, what we do is create a `Query` with the right arguments and then send the query off to be handled by whoever is subscribed to `Game.Queries`. Every single listening component gets a chance to modify the baseline `attack` value.

So let's now implement the modifiers. Once again, we'll make a base class, but this time round it won't have a body for the `Handle()` method:

```
public abstract class CreatureModifier : IDisposable
{
  protected Game game;
  protected Creature creature;

  protected CreatureModifier(Game game, Creature creature)
  {
    this.game = game;
    this.creature = creature;
    game.Queries += Handle; // subscribe
  }

  protected abstract void Handle(object sender, Query q);

  public void Dispose()
  {
    game.Queries -= Handle; // unsubscribe
  }
}
```

Right, so this time round the `CreatureModifier` class is even more sophisticated. It obviously keeps a reference to the creature it's meant to modify, but also to the `Game` that's being played. Why? Well, as you can see, what's happening is that, in the constructor, it subscribes to the `Queries` event so that its inheritors can inject themselves as a set of modifiers is applied one after another. We also implement `IDisposable` so as to unsubscribe from the query events and prevent memory leaks.[29]

The `CreatureModifier.Handle()` method is deliberately made abstract so that inheritors can implement it and handle the modification process depending

[29]This is precisely what is done in Reactive Extensions. See the *Memento* chapter for more information.

on the Query that's being sent. Let's take a look at how this is used by reimplementing DoubleCreatureModifier in this new paradigm:

```
public class DoubleAttackModifier : CreatureModifier
{
  public DoubleAttackModifier(Game game, Creature creature)
    : base(game, creature) {}

  protected override void Handle(object sender, Query q)
  {
    if (q.CreatureName == creature.Name &&
        q.WhatToQuery == Query.Argument.Attack)
      q.Value *= 2;
  }
}
```

Right, so now we have a concrete implementation of Handle(). Extra care needs to be taken here to identify that the query is, in fact, a query that we want to process. Since a DoubleAttackModifier only cares about queries for an attack value, we verify this particular argument (WhatToQuery) and also make sure that the query is related to the creature we're meant to investigate.

If we now add an IncreaseDefenseModifier (increases defense by 2; implementation omitted) we can now run the following scenario:

```
var game = new Game();
var goblin = new Creature(game, "Strong Goblin", 2, 2);
WriteLine(goblin); // Name: Strong Goblin, attack: 2, defense: 2

using (new DoubleAttackModifier(game, goblin))
{
  WriteLine(goblin); // Name: Strong Goblin, attack: 4, defense: 2
  using (new IncreaseDefenseModifier(game, goblin))
  {
    WriteLine(goblin); // Name: Strong Goblin, attack: 4, defense: 4
  }
}

WriteLine(goblin); // Name: Strong Goblin, attack: 2, defense: 2
```

What's happening here? Well, prior to being modified, the goblin is a 2/2. Then, we manufacture a scope, within which the goblin is affected by a `DoubleAttack-Modifier`, so inside the scope, it is a 4/2 creature. As soon as we exit the scope, the modifier's destructor triggers and it disconnects itself from the broker and thus no longer affects the values when they are queried. Consequently, the goblin itself reverts to being a 2/2 creature once again.

Summary

Chain of Responsibility is a very simple design pattern that lets components process a command (or a query) in turn. The simplest implementation of CoR is one where you simply make a reference chain and, in theory, you could replace it with just an ordinary `List` or, perhaps, a `LinkedList` if you wanted fast removal as well.

A more sophisticated Broker Chain implementation that also leverages Mediator and Observer patterns allows us to process queries on an event, letting each subscriber perform modifications of the originally passed object (it's a single reference that goes through the entire chain) before the final values are returned to the client.

Command

Think about a trivial variable assignment, such as `meaningOfLife = 42`. The variable got assigned, but there's no record anywhere that the assignment took place. Nobody can give us the previous value. We cannot take the *fact* of assignment and serialize it somewhere. This is problematic, because without a record of the change, we are unable to roll back to previous values, perform audits or do history-based debugging.[30]

The Command design pattern proposes that, instead of working with objects directly by manipulating them through their APIs, we send them *commands*: instructions on how to do something. A command is nothing more than a data class with its members describing what to do and how to do it. Let's take a look at a typical scenario.

Scenario

Let's try to model a typical bank account that has a balance and an overdraft limit. We'll implement `Deposit()` and `Withdraw()` method on it:

```
public class BankAccount
{
  private int balance;
  private int overdraftLimit = -500;

  public void Deposit(int amount)
  {
    balance += amount;
    WriteLine($"Deposited ${amount}, balance is now {balance}");
  }

  public void Withdraw(int amount)
```

[30] We *do* have dedicated historical debugging tools such as Visual Studio's IntelliTrace or UndoDB.

```
  {
    if (balance - amount >= overdraftLimit)
    {
      balance -= amount;
      WriteLine($"Withdrew ${amount}, balance is now {balance}");
    }
  }

  public override string ToString()
  {
    return $"{nameof(balance)}: {balance}";
  }
}
```

Now we can call the methods directly, of course, but let us suppose that, for audit purposes, we need to make a record of every deposit and withdrawal made and we cannot do it right inside `BankAccount` because – guess what – we've already designed, implemented and tested that class.[31]

Implementing the Command Pattern

We'll begin by defining an interface for a command.

```
public interface ICommand
{
  void Call();
}
```

Having made the interface, we can now use it to define a `BankAccountCommand` that will encapsulate information about what to do with a bank account:

[31]You *can* design your code in a Command-first fashion, i.e. ensure that commands are the only publicly accessible API that your objects provide.

```
public class BankAccountCommand : ICommand
{
  private BankAccount account;
  public enum Action
  {
    Deposit, Withdraw
  }
  private Action action;
  private int amount;

  public BankAccountCommand
    (BankAccount account, Action action, int amount) { ... }
}
```

The information contained in the command includes the following:

- The account to operate upon.
- The action to take; both the set of options and the variable to store the action are defined in the class.
- The amount to deposit or withdraw.

Once the client provides this information, we can take it and use it to perform the deposit or withdrawal:

```
public void Call()
{
  switch (action)
  {
    case Action.Deposit:
      account.Deposit(amount);
      succeeded = true;
      break;
    case Action.Withdraw:
      succeeded = account.Withdraw(amount);
      break;
    default:
      throw new ArgumentOutOfRangeException();
  }
}
```

With this approach, we can create the command and then perform modifications of the account right on the command:

```
var ba = new BankAccount();
var cmd = new BankAccountCommand(ba,
  BankAccountCommand.Action.Deposit, 100);
cmd.Call(); // Deposited $100, balance is now 100
WriteLine(ba); // balance: 100
```

This will deposit 100 dollars into our account. Easy! And if you're worried that we're still exposing the original `Deposit()` and `Withdraw()` member functions to the client, well, the only way to hide them is to make commands inner classes of the `BankAccount` itself.

Undo Operations

Since a command encapsulates all information about some modification to a `BankAccount`, it can equally roll back this modification and return its target object to its prior state.

To begin with, we need to decide whether to stick undo-related operations into our Command interface. I will do it here for purposes of brevity, but in general, this is a design decision that needs to respect the Interface Segregation Principle that we discussed at the beginning of the book. For example, if you envisage some commands being final and not subject to undo mechanics, it might make sense to split `ICommand` into, say, `ICallable` and `IUndoable`.

Anyways, here's the updated `ICommand`:

```
public interface ICommand
{
  void Call();
  void Undo();
}
```

And here is a naive (but working) implementation of `BankAccountCommand.Undo()`, motivated by the (incorrect) assumption that `Deposit()` and `Withdraw()` are symmetric operations:

```
public void Undo()
{
  switch (action)
  {
    case Action.Deposit:
      account.Withdraw(amount);
      break;
    case Action.Withdraw:
      account.Deposit(amount);
      break;
    default:
      throw new ArgumentOutOfRangeException();
  }
}
```

Why is this implementation broken? Because if you tried to withdraw an amount equal to the GDP of a developed nation, you would not be successful, but when rolling back the transaction, we don't have a way of telling that it failed!

To get this information, we modify `withdraw()` to return a success flag:

```
public bool Withdraw(int amount)
{
  if (balance - amount >= overdraftLimit)
  {
    balance -= amount;
    Console.WriteLine($"Withdrew ${amount}, balance is now {balance}");
    return true; // succeeded
  }
  return false; // failed
}
```

That's much better! We can now modify the entire `BankAccountCommand` to do two things:

- Store internally a `succeeded` flag when a withdrawal is made. We assume that `Deposit()` cannot fail.
- Use this flag when `Undo()` is called.

Here we go:

```
public class BankAccountCommand : ICommand
{
  ...
  private bool succeeded;
}
```

Okay, so now we have the flag, we can improve our implementation of `Undo()`:

```
public void Undo()
{
  if (!succeeded) return;
  switch (action)
  {
    case Action.Deposit:
      account.Deposit(amount); // assumed to always succeed
      succeeded = true;
      break;
    case Action.Withdraw:
      succeeded = account.Withdraw(amount);
      break;
    default:
      throw new ArgumentOutOfRangeException();
  }
}
```

Tada! We can finally undo withdrawal commands in a consistent fashion.

```
var ba = new BankAccount();
var cmdDeposit = new BankAccountCommand(ba,
  BankAccountCommand.Action.Deposit, 100);
var cmdWithdraw = new BankAccountCommand(ba,
  BankAccountCommand.Action.Withdraw, 1000);
cmdDeposit.Call();
cmdWithdraw.Call();
WriteLine(ba); // balance: 100
cmdWithdraw.Undo();
cmdDeposit.Undo();
WriteLine(ba); // balance: 0
```

The goal of this exercise was, of course, to illustrate that in addition to storing information about the action to perform, a Command can also store some intermediate information that is, once again, useful for things like audits. If you detect a series of 100 failed withdrawal attempts, you can investigate a potential hack.

Composite Commands (a.k.a. Macros)

A transfer of money from account A to account B can be simulated with two commands:

1. Withdraw $X from A
2. Deposit $X to B

It would be nice if, instead of creating and calling these two commands, we could just create and call a single command that encapsulates both of the above. This is the essence of the Composite design pattern that we'll discuss later.

Let's define a skeleton composite command. I'm going to inherit from `List<BankAccountCo` and, of course, implement the `ICommand` interface:

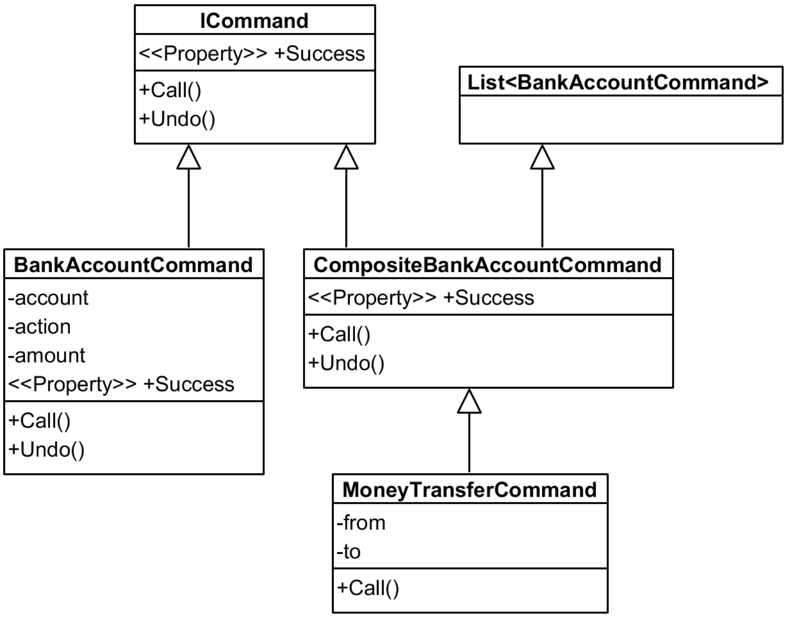
```
abstract class CompositeBankAccountCommand : List<BankAccountCommand>, ICommand
{
  public virtual void Call()
  {
    ForEach(cmd => cmd.Call());
  }

  public virtual void Undo()
  {
    foreach (var cmd in
      ((IEnumerable<BankAccountCommand>)this).Reverse())
    {
      cmd.Undo();
    }
  }
}
```

As you can see, the CompositeBankAccountCommand is both a list as well as an Command, which fits the definition of the Composite design pattern. I've implemented both Undo() and Redo() operations; note that the Undo() process goes through commands in reverse order; hopefully I don't have to explain *why* you'd want this as default behavior. The cast is there because a List<T> has its own, void-returning, mutating Reverse() that we definitely do not want. If you don't like what you see here, you can use a for loop or some other base type that doesn't do in-place reversal.

So now, how about a composite command specifically for transferring money? I would define it as follows:

```
class MoneyTransferCommand : CompositeBankAccountCommand
{
  public MoneyTransferCommand(BankAccount from,
    BankAccount to, int amount)
  {
    AddRange(new []
    {
      new BankAccountCommand(from,
        BankAccountCommand.Action.Withdraw, amount),
      new BankAccountCommand(to,
        BankAccountCommand.Action.Deposit, amount)
    });
  }
}
```

As you can see, all we're doing is providing a constructor to initialize the object with. We keep reusing the base class Undo() and Redo() implementations.



But wait, that's not right, is it? The base class implementations don't quite cut it because they don't incorporate the idea of failure. If I fail to withdraw money from A, I shouldn't deposit that money to B: the entire chain should cancel itself.

To support this idea, more drastic changes are required. We need to

- Add a `Success` flag to `Command`. This of course implies that we can no longer use an interface - we need an abstract class.
- Record the success or failure of *every* operation.
- Ensure that the command can only be undone if it originally succeeded.
- Introduce a new in-between class called `DependentCompositeCommand` that is very careful about actually rolling back the commands.

Let's assume that we've performed the refactoring such that `Command` is now an abstract class with a Boolean `Success` member; the `BankAccountCommand` now `overrides` both `Undo()` and `Redo()`.

When calling each command, we only do so if the previous one succeeded; otherwise we simply set the `success` flag to `false`.

```
public override void Call()
{
  bool ok = true;
  foreach (var cmd in this)
  {
    if (ok)
    {
      cmd.Call();
      ok = cmd.Success;
    }
    else
    {
      cmd.Success = false;
    }
  }
}
```

There is no need to override the `Undo()` because each of our commands checks its own `Success` flag and undoes the operation only if it's set to `true`. Here's a scenario that demonstrates the correct operation of the new scheme when the source account doesn't have enough funds for the transfer to succeed.

```
var from = new BankAccount();
from.Deposit(100);
var to = new BankAccount();

var mtc = new MoneyTransferCommand(from, to, 1000);
mtc.Call();
WriteLine(from); // balance: 100
WriteLine(to);  // balance: 0
```

One can imagine an even stronger form of the above where a composite command only succeeds if *all* of its parts succeed (think about a transfer where the withdrawal succeeds but the deposit fails because the account is locked - would you want it to go through?) – this is a bit harder to implement, and I leave it as an exercise for the reader.

The entire purpose of this section was to illustrate how a simple Command-based approach can get quite complicated when real-world business requirements are taken into account. Whether or not you actually *need* this complexity… well, that is up to you.

Functional Command

The Command design pattern is typically implemented using classes. It is, however, possible to also implement this pattern in a functional way.

First of all, one might argue that an `ICommand` interface with a single `Call()` method is simply unnecessary: we already have delegates such as `Func` and `Action` that can serve as *de facto* interfaces for our purposes. Similarly, when it comes to invoking the commands, we can invoke said delegates directly instead of calling a member of some interface.

Here's a trivial illustration of the approach. We begin by defining a `BankAccount` simply as:

```
public class BankAccount
{
  public int Balance;
}
```

We can then define different commands to operate on the bank account as independent methods. These could, alternatively, be packaged into ready-made function objects: there's no real difference between the two:

```
public void Deposit(BankAccount account, int amount)
{
  account.Balance += amount;
}

public void Withdraw(BankAccount account, int amount)
{
  if (account.Balance >= amount)
    account.Balance -= amount;
}
```

Every single method represents a command. We can therefore bundle up the commands in a simple list and process them one after another:

```
var ba = new BankAccount();
var commands = new List<Action>();

commands.Add(() => Deposit(ba, 100));
commands.Add(() => Withdraw(ba, 100));

commands.ForEach(c => c());
```

You may feel that this model is a great simplification of the one we had previously when talking about `ICommand`. After all, any invocation can be reduced to a parameterless `Action` that simply captures the needed elements in the lambda. However, this approach has significant downsides, namely:

- **Direct references**: a lambda that captures a specific object by necessity extends its lifetime. While this is great in terms of correctness (you'll never invoke a command with a nonexistent object) there are situations where you want commands to persist longer than the objects they need to affect.
- **Logging**: if you wanted to record every single action being performed on an account, you *still* need some sort of command processor. But how can you determine which command is being invoked? All you're looking at is an `Action` or similarly nondescript delegate, how do you determine whether it's a deposit or a withdrawal or something entirely different, like a composite command?
- **Marshaling**: quite simply, you cannot marshal a lambda. You could maybe marshal an expression tree (as in, an `Expression<Func<>>`) but even then, parsing expression trees is not the easiest of things. A conventional OOP-based approach is easier because a class can be deterministically (de)serialized.
- **Secondary operations**: unlike functional objects, an OOP command (or its interface) can define operations other than invocation. We've looked at examples such as `Undo()` but other operations could include things like `Log()`, `Print()` or something else. A functional approach doesn't give you this sort of flexibility.

To sum up, while the functional pattern does represent some action that needs to be done, it only encapsulates its principal behavior. A function is difficult to inspect/traverse, it is difficult to serialize, and if it captures context this has obvious lifetime implications. Use with caution!

Queries and Command Query Separation

The notion of Command Query Separation (CQS) is the idea that operations in a system fall broadly into the following two categories:

- Commands, which are instructions for the system to perform some operation that involves mutation of state, but yields no value; and
- Queries, which are requests for information that yield values but do not mutate state.

The GoF book does not define a Query as a separate pattern, so in order to settle this issue once and for all, I propose the following, very simple, definition:

A *query* is a special type of command that does not mutate state. Instead, a query instructs components to provide some information, such as a value calculated on the basis of interaction with one or more components.

There. We can now argue that both parts of CQS fall under the Command design pattern, the only difference being that queries have a return value – not in the `return` sense, of course, but rather in having a mutable field/property that any command processor can initialize or modify.

Summary

The Command design pattern is simple: what it basically suggests is that components can communicate with one another using special objects that encapsulate instructions, rather than specifying those same instructions as arguments to a method.

Sometimes, you don't want such an object to mutate the target or cause it to do something specific; instead you want to use such an object to get some info from the target, in which case we typically call such an object a Query. While, in most cases, a query is an immutable object that relies on the return type of the method, there *are* situations (see, e.g., the Chain of Responsibility “Broker Chain” example) when you want the result that’s being returned to be modified by other components. But the components themselves are still not modified, only the result is.

Commands are used a lot in UI systems to encapsulate typical actions (e.g., Copy or Paste) and then allow a single command to be invoked by several different means. For example, you can Copy by using the top-level application menu, a button on the toolbar, the context menu, or by pressing a keyboard shortcut.

Finally, these actions can be combined into composite commands (macros) – sequences of actions that can be recorded and then replayed at will. Notice that a composite command can also be composed of other composite command (as per the Composite design pattern).

Interpreter

> Any good software engineer will tell you that a compiler and an interpreter are interchangeable. — *Tim Berners-Lee.*

The goal of the Interpreter design pattern is, you guessed it, to interpret input, particularly *textual* input, although to be fair it really doesn't matter. The notion of an Interpreter is greatly linked to Compiler Theory and similar courses taught at universities. Since we don't have nearly enough space here to delve into the complexities of different types of parsers and whatnot, the purpose of this chapter is to simply show some examples of the kinds of things you might want to interpret.

Here are a few fairly obvious ones:

- Numeric literals such as `42` or `1.234e12` need to be interpreted to be stored efficiently in binary. In C#, these operations are covered via methods such as `Int.Parse()`.[32]
- Regular expressions help us find patterns in text, but what you need to realize is that regular expressions are essentially a separate, embedded domain-specific language (DSL). And naturally, before using them, they must be interpreted correctly.
- Any structured data, be it CSV, XML, JSON or something more complicated, requires interpretation before it can be used.
- At the pinnacle of the application of Interpreter, we have fully fledged programming languages. After all, a compiler or interpreter for a language like C or Python must actually understand the language before compiling something executable.

Given the proliferation and diversity of challenges related to interpretation, we shall simply look at some examples. These serve to illustrate how one can build an Interpreter: making one from scratch or using a specialized library or parser framework.

[32]The parsing of numbers is the #1 operation that gets redefined (optimized) by developers of algorithmic trading systems. The default implementations are very powerful and can handle many different number formats, but in real life the stock market typically feeds you data with uniform precision and notation, allowing the construction of much faster (orders of magnutude) parsers.

Numeric Expression Evaluator

Let’s imagine that we decide to parse *very* simple mathematical expressions such as 3+(5-4), i.e., we’ll restrict ourselves to addition, subtraction and brackets. We want a program that can read such an expression and, of course, calculate the expression’s final value.

We are going to build the calculator *by hand*, without resorting to any parsing framework. This should hopefully highlight *some* of the complexity involved in parsing textual input.

Lexing

The first step to interpreting an expression is called *lexing*, and it involves turning a sequence of characters into a sequence of *tokens*. A token is typically a primitive syntactic element, and we should end up with a flat sequence of these. In our case, a token can be

- An integer
- An operator (plus or minus)
- An opening or closing parenthesis

Thus, we can define the following structure:

```
public class Token
{
  public enum Type
  {
    Integer, Plus, Minus, Lparen, Rparen
  }

  public Type MyType;
  public string Text;

  public Token(Type type, string text)
  {
    MyType = type;
    Text = text;
  }

  public override string ToString()
  {
    return $"`{Text}`";
  }
}
```

You'll note that `Token` is not an `enum` because, apart from the type, we also want to store the text that this token relates to, since it is not always predefined. (We could, alternatively, store some `Range` that would refer to the original string.)

So now, given a `string` containing an expression, we can define a lexing process that will turn a text into a `List<Token>`:

```
static List<Token> Lex(string input)
{
  var result = new List<Token>();

  for (int i = 0; i < input.Length; i++)
  {
    switch (input[i])
    {
      case '+':
        result.Add(new Token(Token.Type.Plus, "+"));
        break;
      case '-':
        result.Add(new Token(Token.Type.Minus, "-"));
        break;
      case '(':
        result.Add(new Token(Token.Type.Lparen, "("));
        break;
      case ')':
        result.Add(new Token(Token.Type.Rparen, ")"));
        break;
      default:
        // todo
    }
  }

  return result;
}
```

Parsing predefined tokens is easy. In fact, we could have added them as a

```
Dictionary<BinaryOperation.Type, char>
```

to simplify things. But parsing a number is not so easy. If we hit a `1`, we should wait and see what the next character is. For this we define a separate routine:

```
var sb = new StringBuilder(input[i].ToString());
for (int j = i + 1; j < input.Length; ++j)
{
  if (char.IsDigit(input[j]))
  {
    sb.Append(input[j]);
    ++i;
  }
  else
  {
    result.Add(new Token(Token.Type.Integer, sb.ToString()));
    break;
  }
}
```

Essentially, while we keep reading (pumping) digits, we add them to the buffer. When we're done, we make a `Token` out of the entire buffer and add it to the resulting list.

Parsing

The process of *parsing* turns a sequence of tokens into meaningful, typically object-oriented, structures. At the top, it's often useful to have an abstract class or interface that all elements of the tree implement:

```
public interface IElement
{
  int Value { get; }
}
```

The type’s `Value` evaluates this element’s numeric value. Next, we can create an element for storing integral values (such as 1, 5 or 42):

```
public class Integer : IElement
{
  public Integer(int value)
  {
    Value = value;
  }

  public int Value { get; }
}
```

If we don’t have an `Integer`, we must have an operation such as addition or subtraction. In our case, all operations are *binary*, meaning they have two parts. For example, 2+3 in our model can be represented in pseudocode as `BinaryOperation{Literal{2}, Literal{3}, addition}`:

```
public class BinaryOperation : IElement
{
  public enum Type
  {
    Addition,
    Subtraction
  }

  public Type MyType;
  public IElement Left, Right;

  public int Value
  {
    get
    {
      switch (MyType)
      {
        case Type.Addition:
          return Left.Value + Right.Value;
        case Type.Subtraction:
          return Left.Value - Right.Value;
        default:
          throw new ArgumentOutOfRangeException();
      }
    }
  }
}
```

But anyways, on to the parsing process. All we need to do is turn a sequence of Tokens into a binary tree of IExpressions. From the outset, it can look as follows:

```
static IElement Parse(IReadOnlyList<Token> tokens)
{
  var result = new BinaryOperation();
  bool haveLHS = false;
  for (int i = 0; i < tokens.Count; i++)
  {
    var token = tokens[i];

    // look at the type of token
    switch (token.MyType)
    {
      // process each token in turn
    }
  }
  return result;
}
```

The only thing we need to discuss from the above code is the haveLHS variable. Remember, what are are trying to get is a tree, and at the *root* of that tree we expect a BinaryExpression which, by definition, has left and right sides. But when we are on a number, how do we know if it's the left or right side of an expression? That's right, we don't, which is why we track this using haveLHS.

Now let's go through these case by case. First, integers – these map directly to our `Integer` construct, so all we have to do is turn text into a number. (Incidentally, we could have also done this at the lexing stage if we wanted to.)

```
case Token.Type.Integer:
  var integer = new Integer(int.Parse(token.Text));
  if (!haveLHS)
  {
    result.Left = integer;
    haveLHS = true;
  } else
  {
    result.Right = integer;
  }
  break;
```

The `plus` and `minus` tokens simply determine the type of the operation we're currently processing, so they're easy:

```
case Token.Type.Plus:
  result.MyType = BinaryOperation.Type.Addition;
  break;
case Token.Type.Minus:
  result.MyType = BinaryOperation.Type.Subtraction;
  break;
```

And then there's the left parenthesis. Yep, just the left, we don't detect the right one explicitly. Basically, the idea here is simple: find the closing right parenthesis (I'm ignoring nested brackets for now), rip out the entire subexpression, `Parse()` it recursively and set as the left or right-hand side of the expression we're currently working with:

```
case Token.Type.Lparen:
  int j = i;
  for (; j < tokens.Count; ++j)
    if (tokens[j].MyType == Token.Type.Rparen)
      break; // found it!
  // process subexpression w/o opening
  var subexpression = tokens.Skip(i+1).Take(j - i - 1).ToList();
  var element = Parse(subexpression);
  if (!haveLHS)
  {
    result.Left = element;
    haveLHS = true;
  } else result.Right = element;
  i = j; // advance
  break;
```

In a real-world scenario, you'd want a lot more safety features in here: not just handling nested parentheses (which I think is a must), but handling incorrect expressions where the closing parenthesis is missing. If it is indeed missing, how would you handle it? Throw an exception? Try to parse whatever's left and assume the closing is at the very end? Something else? All of these issues are left as the exercise to the reader.

Using Lexer and Parser

With both `Lex()` and `Parse()` implemented, we can finally parse the expression and calculate its value:

```
var input = "(13+4)-(12+1)";
var tokens = Lex(input);
WriteLine(string.Join("\t", tokens));
// `(`  `13`  `+`  `4`  `)`  `-`  `(`  `12`  `+`  `1`  `)`

var parsed = Parse(tokens);
WriteLine($"{input} = {parsed.Value}");
// (13-4)-(12+1) = -4
```

Interpretation in the Functional Paradigm

If you look at a set of elements that are produced by either the lexing or the parsing process, you will quickly see that they are trivial structures that would map very neatly onto F#'s discriminated unions. This, in turn, allows us to subsequently use pattern matching when there comes a time to traverse a (recursive) discriminated union in order to transform it into something else.

Here's an example: suppose you are given a definition of a mathematical expression and you want to print or evaluate it[33]. Let's define the structure in XML so we don't have to go through a difficult parsing process:

```
<math>
  <plus>
    <value>2</value>
    <value>3</value>
  </plus>
</math>
```

We can create a recursive discriminated union to represent this structure:

```
type Expression =
  Math of Expression list
  | Plus of lhs:Expression * rhs:Expression
  | Value of value:string
```

As you can see, there is a 1-to-1 correspondence between the XML elements and the corresponding `Expression` cases (e.g., `<math>` ⊠ `Math`). In order to instantiate cases, we would need to use reflection. One trick I adopt here is to precompute the case constructors using APIs from the `Microsoft.FSharp.Reflection` namespace:

[33]This is a small illustration of something that's a real-life commercial product called MathSharp — a tool that converts MathML notation to ready-to-compile code. See http://activemesa.net/mathsharp for more information.

```
let cases = FSharpType.GetUnionCases (typeof<Expression>)
            |> Array.map(fun f ->
              (f.Name, FSharpValue.PreComputeUnionConstructor(f)))
            |> Map.ofArray
```

We can then write a function that constructs a union case given a name and a set of parameters:

```
let makeCase parameters =
    try
      let caseInfo = cases.Item name
      (caseInfo parameters) :?> Expression
    with
    | exp -> raise <| new Exception(String.Format("Failed to create {0} : {1}",\
 name, exp.Message))
```

In the listing above, the variable `name` is captured implicitly, since the `makeCase` function is an inner function. But let's not jump ahead. What we're interested in is, of course, parsing and transforming some piece of XML. Here's how that process would begin:

```
use stringReader = new StringReader(text)
use xmlReader = XmlReader.Create(stringReader)
let doc = XDocument.Load(xmlReader)
let parsed = recursiveBuild doc.Root
```

So, what is this `recursiveBuild` function? As the name it suggests, it's a function that recursively turns an XML element into a case of our discriminated union. Here is the full listing:

```
let rec recursiveBuild (root:XElement) =
  let name = root.Name.LocalName |> makeCamelCase

  let makeCase parameters =
    // as before

  let elems = root.Elements() |> Seq.toArray
  let values = elems |> Array.map(fun f -> recursiveBuild f)
  if elems.Length = 0 then
    let rootValue = root.Value.Trim()
    makeCase [| box rootValue |]
  else
    try
      values |> Array.map box |> makeCase
    with
    | _ -> makeCase [| values |> Array.toList |]
```

Let's try to go slowly through what's going on here:

- Since our union cases are camel-cased and the XML file is lowercase, I convert the name of the XML element (which we call `root`) to camel case.
- We materialize the sequence of child elements of the current elements into an array.
- For each inner element, we call `recursiveBuild` recursively (surprise!).
- Now we check how many child elements the current element has. If it's zero, it could be just a `<value>` with text in it. If it's not, there are two possibilities:
  - The item takes a bunch of primitives that can all be boxed into parameters.
  - The item takes a bunch of expressions.

This constructs the expression tree. If we want to evaluate the numeric value of the expression, this is now simple thanks to pattern matching:

```
let rec eval expr =
  match expr with
  | Math m -> eval m.Head
  | Plus (lhs, rhs) -> eval lhs + eval rhs
  | Value v -> v |> int
```

Similarly, you could define a function for printing an expression:

```
let rec print expr =
  match expr with
  | Math m -> print m.Head
  | Plus (lhs, rhs) -> String.Format("({0}+{1})", print lhs, print rhs)
  | Value v -> v
```

Putting it all together, we can now print the expression in human-readable form and evaluate its result:

```
let parsed = recursiveBuild doc.Root
printf "%s = %d" (print parsed) (eval parsed)
// (2+3) = 5
```

Both of the functions are, of course, crude implementations of the Visitor design pattern without any traditional OOP trappings (though they are, of course, present behind the scenes). Some things to note are:

- Our `Value` case is `of string`. If we wanted it to store an integer or a floating-point number, our parsing code would have to pry this information away using reflection.
- Instead of making top-level functions, we can give `Expression` its own methods and even properties. For example, we can give it a property called `Val` that evaluates its numeric value:

```
type Expression =
  // union members here
  member self.Val =
    let rec eval expr =
      match expr with
      | Math m -> eval(m.Head)
      | Plus (lhs, rhs) -> eval lhs + eval rhs
      | Value v -> v |> int
    eval self
```

- Strictly speaking, discriminated unions violate the Open-Closed Principle since there is no way to augment them through inheritance. As a result, if you decide to support new cases, you'd have to modify the original union type.

To sum up, discriminated unions, pattern matching and also list comprehensions (which we haven't used in our demo, but you'd typically use them in a scenario like this) all make the Interpreter and Visitor patterns easy to implement under the functional paradigm.

Summary

First of all, it needs to be said that, comparatively speaking, the Interpreter design pattern is somewhat uncommon – the challenges of building parsers is nowadays considered inessential, which is why we see it being removed from Computer Science courses in many universities (my own one included). Also, unless you plan to work in language design or, say, making tools for static code analysis, you are unlikely to find the skills in building parsers in high demand.

That said, the challenge of interpretation is a whole separate field of Computer Science that a single chapter of a Design Patterns book cannot reasonably do justice to. If you are interested in the subject, I recommend you check out frameworks such as Lex/Yacc, ANTLR and many others that are specifically geared for lexer/parser construction. I can also recommend writing static analysis plugins for popular IDEs – this is a great way to get a feel for how real ASTs look, how they are traversed and even modified.

Iterator

An iterator, put simply, is an object that is used to traverse some structure or other. Typically, the iterator references the currently accessed element and has a method to move forward. A bidirectional iterator also lets you lets you walk backwards, and a random-access iterator allows you to access an element at an arbitrary position.

In .NET, that thing which enables iterator typically implements the `IEnumerator<T>` interface. It has the following members:

- `Current` refers to the element at the current position.
- `MoveNext()` lets you move on to the next element of the collection; returns `true` if we succeeded and `false` otherwise.
- `Reset()` sets the enumerator to the initial position.

The enumerator is also disposable, but we don't care about that too much. The point is, any time you write

```
foreach (x in y)
  Console.WriteLine(x);
```

what you're really doing is the equivalent of

```
var enumerator = ((IEnumerable<Foo>)y).GetEnumerator();
while (enumerator.MoveNext())
{
  temp = enumerator.Current;
  Console.WriteLine(temp);
}
```

In other words, a class that implements `IEnumerable<T>` is required to have a method called `GetEnumerator()` that returns an `IEnumerator<T>`. And you use that enumerator to traverse the object.

Needless to say, it is very rare for you to have to make your own `IEnumerator`. Typically, you can write code such as…

```
IEnumerable<int> GetSomeNumbers()
{
  yield return 1;
  yield return 2;
  yield return 3;
}
```

…and the rest of the operations will be taken care of by the compiler. Alternatively, you can just use an existing collection class (array, `List<T>`, etc.) which already has all the plumbing you need.

Array-Backed Properties

Not all things are easy to iterate. For example, you cannot iterate all fields in a class unless you're using reflection. But sometimes you need to. Here, let me show you a scenario.

Suppose you're making a game with creatures in it. These creatures have various attributes, such as strength, agility and intelligence. You could implement them as

```
public class Creature
{
  public int Strength { get; set; }
  public int Agility { get; set; }
  public int Intelligence { get; set; }
}
```

But now you also want to output some aggregate statistics about the creature. For example, you decide to calculate the sum of all its abilities:

```
public double SumOfStats => Strength + Agility + Intelligence;
```

This code is impossible to automatically refactor if you add an additional `Wisdom` property (ooh, is that too much D&D nerdiness for you?), but let me show you something even worse. If you want the average of all the abilities, you would write:

```
public double AverageStat => SumOfStats / 3.0;
```

Whoa there! That 3.0 is a bona fide *magic number*, completely unsafe if the structure of the code changes. Let me show you yet another example of ugliness. Suppose you decide to calculate the maximum ability value of a creature. You'd need to write something like:

```
public double MaxStat => Math.Max(
  Math.Max(Strength, Agility), Intelligence);
```

Well, you get the idea. This code is not robust and will break on any small change, so we're going to fix it, and the implementation is going to make use of *array-backed properties*.

The idea of array-backed properties is simple: all the backing fields of related properties exist in one array:

```
private int [] stats = new int[3];
```

Each of the properties then projects its getter and setter into the array. To avoid using integral indices you can introduce private constants:

```
private const int strength = 0;
public int Strength
{
  get => stats[strength];
  set => stats[strength] = value;
}
// same for other properties
```

And now, of course, calculating the sum/average/maximum statistics is really easy because the underlying field is an array, and arrays are supported in LINQ:

```
public double AverageStat => stats.Average();
public double SumOfStats => stats.Sum();
public double MaxStat => stats.Max();
```

If you want to add an extra property, all you need to do is:

- Extend the array by one element
- Create a property with getter and setter

And that's it! The stats will still be calculated correctly. Furthermore, if you want, you can eschew all of those methods we made in favor of

```
public IEnumerable<int> Stats => stats;
```

and just let the client perform his own LINQ queries directly, e.g., `creature.Stats.Averag`

Finally, if you want `stats` to be the enumerable collection, i.e. letting people write `foreach (var stat in creature)`, you can simply implement `IEnumerable` (and perhaps an indexer too):

```
public class Creature : IEnumerable<int>
{
  // as before

  public IEnumerator<int> GetEnumerator()
    => stats.AsEnumerable().GetEnumerator();

  IEnumerator IEnumerable.GetEnumerator()
    => GetEnumerator();

  public int this[int index]
  {
    get => stats[index];
    set => stats[index] = value;
  }
}
```

This approach is functional, but there are plenty of downsides. One of those downsides has to do with change notifications. For example, suppose that your UI application binds a UI element to the `SumOfStats` property. You change `Strength`, but how would `SumOfStats` let you know that it did, in fact, change too? If `SumOfStats` was defined as a basic sum of different properties, we could have treated that summation as an expression tree, parsed it and extracted the dependencies. But because we're using LINQ, this is now impossible or, at the very least, very difficult. We could attempt to supply some special metadata to indicate that some properties are array-backed, and then read this metadata when determining dependencies but, as you can guess, this has both computational and cognitive costs.

Let's Make an Iterator

In order to appreciate just how ugly iterators can get if you do decide to make them directly, we are going to implement a classic Comp Sci example: tree traversal. Let's begin by defining a single node of a binary tree:

```
public class Node<T>
{
  public T Value;
  public Node<T> Left, Right;
  public Node<T> Parent;

  public Node(T value)
  {
    Value = value;
  }

  public Node(T value, Node<T> left, Node<T> right)
  {
    Value = value;
    Left = left;
    Right = right;

    left.Parent = right.Parent = this;
  }
}
```

I've thrown in an additional constructor that initializes its node with both left and right child nodes. This allows us to define chained constructor trees such as:

```
//    1
//   / \
// 2    3
var root = new Node<int>(1,
  new Node<int>(2), new Node<int>(3));
```

Okay, so now we want to traverse the tree. If you remember your Data Structures and Algorithms course, you'll know that there are three ways: in-order, preorder and postorder. Suppose we decide to define an InOrderIterator. Here's what it would look like:

```
public class InOrderIterator<T>
{
  public Node<T> Current { get; set; }
  private readonly Node<T> root;
  private bool yieldedStart;

  public InOrderIterator(Node<T> root)
  {
    this.root = Current = root;
    while (Current.Left != null)
      Current = Current.Left;
  }

  public bool MoveNext()
  {
    // todo
  }
}
```

Not bad so far: just as if we were implementing `IEnumerator<T>`, we have a property called `Current` and a `MoveNext()` method. But here's the thing: since the iterator is stateful, every invocation of MoveNext() has to take us to the next element in our current traversal scheme. This isn't as easy as it sounds:

```
public bool MoveNext()
{
  if (!yieldedStart)
  {
    yieldedStart = true;
    return true;
  }

  if (Current.Right != null)
  {
    Current = Current.Right;
    while (Current.Left != null)
      Current = Current.Left;
    return true;
  }
  else
  {
    var p = Current.Parent;
    while (p != null && Current == p.Right)
    {
      Current = p;
      p = p.Parent;
    }
    Current = p;
    return Current != null;
  }
}
```

Whoa there! Bet you didn't expect this! Well this is exactly what you get if you implement your own iterators directly: an unreadable mess. But it works! We can use the iterator directly, C++ style:

```
var it = new InOrderIterator<int>(root);
while (it.MoveNext())
{
  Write(it.Current.Value);
  Write(',');
}
WriteLine();
// prints 213
```

Or, if we want, we can construct a dedicated `BinaryTree` class that exposes this in-order iterator as a default one:

```
public class BinaryTree<T>
{
  private Node<T> root;

  public BinaryTree(Node<T> root)
  {
    this.root = root;
  }

  public InOrderIterator<T> GetEnumerator()
  {
    return new InOrderIterator<T>(root);
  }
}
```

Notice we don't even have to implement `IEnumerable` (thanks to duck typing[34]). We can now write:

[34]*Duck typing* is the idea that 'if it walks like a duck and it quacks like a duck, it is a duck'. In programming parlance, duck typing implies that the right code will be used even when it doesn't implement any particular interface to identify it. In our case, the `foreach` keyword doesn't care in the least whether your type implements `IEnumerable` or not – all it's looking for is the implementation of `GetEnumerator()` in the iterated class. If it finds it, everything works.

```
var root = new Node<int>(1,
  new Node<int>(2), new Node<int>(3));
var tree = new BinaryTree<int>(root);
foreach (var node in tree)
  WriteLine(node.Value); // 2 1 3
```

Improved Iteration

Our implementation of in-order iteration is virtually unreadable and is nothing like what you read in textbooks. Why? Lack of recursion. After all, `MoveNext()` cannot preserve its state, so every time it gets invoked, it starts from scratch without remembering its context: it only remembers the previous element, which needs to be found before we find the next one in the iteration scheme we're using.

And this is why `yield return` exists: you can construct a state machines behind the scenes. This means that if I wanted to create a more natural in-order implementation, I could simply write it as:

```
public IEnumerable<Node<T>> NaturalInOrder
{
  get
  {
    IEnumerable<Node<T>> TraverseInOrder(Node<T> current)
    {
      if (current.Left != null)
      {
        foreach (var left in TraverseInOrder(current.Left))
          yield return left;
      }
      yield return current;
      if (current.Right != null)
      {
        foreach (var right in TraverseInOrder(current.Right))
          yield return right;
      }
    }
    foreach (var node in TraverseInOrder(root))
```

```
      yield return node;
  }
}
```

Notice that all the calls here are recursive. Now what we can do is use this directly, for example:

```
var root = new Node<int>(1,
  new Node<int>(2), new Node<int>(3));
var tree = new BinaryTree<int>(root);
WriteLine(string.Join(",", tree.NaturalInOrder.Select(x => x.Value)));
// 2,1,3
```

Woo-hoo! This is way better. The algorithm itself is readable and, once again, we can take the property and just do LINQ on it, no problem.

Iterator Adapter

Quite often you want an object to be iterable in some special way. For example, suppose you want to calculate the sum of all elements in a matrix – LINQ doesn't provide a Sum() method over a rectangular array, so what you can do is build an adapter such as:

```
public class OneDAdapter<T> : IEnumerable<T>
{
  private readonly T[,] arr;
  private int w, h;

  public OneDAdapter(T[,] arr)
  {
    this.arr = arr;
    w = arr.GetLength(0);
    h = arr.GetLength(1);
  }

  public IEnumerator<T> GetEnumerator()
  {
    for (int y = 0; y < h; ++y)
    for (int x = 0; x < w; ++x)
      yield return arr[x, y];
  }

  IEnumerator IEnumerable.GetEnumerator()
  {
    return GetEnumerator();
  }
}
```

This adapter could then be used whenever you want to iterate a 2D array in 1D manner. For example, calculation of the sum is now as simple as:

```
var data = new [,] { { 1, 2 }, { 3, 4 } };
var sum = new OneDAdapter<int>(data).Sum();
```

Of course, we're still stuck with C#'s inability to derive type arguments in constructors, so perhaps a factory method could be useful here.

Here's another example – this one supports reverse iteration of a 1D array:

```
public class ReverseIterable<T> : IEnumerable<T>
{
  private readonly T[] arr;

  public ReverseIterable(T[] arr) => this.arr = arr;

  public IEnumerator<T> GetEnumerator()
  {
    for (int i = arr.Length - 1; i >= 0; --i)
      yield return arr[i];
  }

  IEnumerator IEnumerable.GetEnumerator()
  {
    return GetEnumerator();
  }
}
```

Again, if you don't want to specify the type parameter explicitly, you'd have to create another, non-generic `ReverseIterable` class and provide a factory method:

```
public static class ReverseIterable
{
  public static ReverseIterable<T> From<T>(T[] arr)
  {
    return new ReverseIterable<T>(arr);
  }
}
```

Of course, as we've discussed countless times before, this implies that the constructor is made public, and the only way to make it private is to make the factory a nested class of the iterator adapter.

Summary

The Iterator design pattern has been deliberately hidden in C# in favor of the simple `IEnumerator`/`IEnumerable` duopoly upon which everything is built. Notice that these interfaces only support forward iteration – there is no `MoveBack()` in `IEnumerator`. The existence of `yield` allows you to very quickly return elements as a collection that can be consumed by soneone else while being blissfully unaware of the state machine that gets built behind the scenes.

Mediator

A large proportion of the code we write has different components (classes) communicating with one another through direct references. However, there are situations when you don't want objects to be necessarily aware of each other's presence. Or, perhaps you *do* want them to be aware of one another, but you still don't want them to communicate through references, because as soon as you keep and hold a reference to something, you extend that object's lifetime beyond what might originally be desired (unless it's a `WeakReference`, of course).

So the Mediator is a mechanism for facilitating communication between the components. Naturally, the mediator itself needs to be accessible to every component taking part, which means it should either be a publicly available static variable or, alternatively, just a reference that gets injected into every component.

Chat Room

Your typical internet chat room is the classic example of the Mediator design pattern, so let's implement this before we move on to the more complicated stuff.

The most trivial implementation of a participant in a chat room can be as simple as

```
public class Person
{
  public string Name;
  public ChatRoom Room;
  private List<string> chatLog = new List<string>();

  public Person(string name) => Name = name;

  public void Receive(string sender, string message)
  {
    string s = $"{sender}: '{message}'";
    WriteLine($"[{Name}'s chat session] {s}");
    chatLog.Add(s);
  }

  public void Say(string message) => Room.Broadcast(Name, message);

  public void PrivateMessage(string who, string message)
  {
    Room.Message(Name, who, message);
  }
}
```

So we've got a person with a `Name` (user id), a chat log and a reference to the actual `ChatRoom`. We have a constructor and then three methods:

- `Receive()` allows us to receive a message. Typically what this function would do is show the message on the user's screen, and also add it to the chat log.
- `Say()` allows the person to broadcast a message to everyone in the room.
- `PrivateMessage()` is private messaging functionality. You need to specify the name of the person the message is intended for.

Both `Say()` and `PrivateMessage()` [35] just relay operations to the chat room. Speaking of which, let's actually implement `ChatRoom` – it's not particularly complicated.

```
public class ChatRoom
{
  private List<Person> people = new List<Person>();

  public void Broadcast(string source, string message) { ... }
  public void Join(Person p) { ... }
  public void Message(string source, string destination,
    string message) { ...  }
}
```

[35]In the real world, I would probably call the method `PM()`, considering how commonplace that acronym has become.

So, I have decided to go with pointers here. The `ChatRoom` API is very simple:

- `Join()` gets a person to join the room. We are not going to implement `Leave()`, instead deferring the idea to a subsequent example in this chapter.
- `Broadcast()` sends the message to everyone… well, not quite everyone: we don't need to send the message back to the person that sent it.
- `Message()` sends a private message.

The implementation of `Join()` is as follows:

```
public void Join(Person p)
{
  string joinMsg = $"{p.Name} joins the chat";
  Broadcast("room", joinMsg);

  p.Room = this;
  people.Add(p);
}
```

Just like a classic IRC chat room, we broadcast the message that someone has joined to everyone in the room. The first argument of `Broadcast()`, the `origin` parameter, in this case, is specified as `"room"` rather than the person that's joined. We then set the person's `room` reference and add them to the list of people in the room.

Now, let's look at `Broadcast()`: this is where a message is sent to every room participant. Remember, each participant has its own `Person.Receive()` method for processing the message, so the implementation is somewhat trivial:

```
public void Broadcast(string source, string message)
{
  foreach (var p in people)
    if (p.Name != source)
      p.Receive(source, message);
}
```

Whether or not we want to prevent a broadcast message to be relayed to ourselves is a point of debate, but I'm actively avoiding it here. Everyone else gets the message, though.

Finally, here's private messaging implemented with `Message()`:

```
public void Message(string source, string destination, string message)
{
  people.FirstOrDefault(p => p.Name == destination)
    ?.Receive(source, message);
}
```

This searches for the recipient in the list of `people` and, if the recipient is found (because who knows, they could have left the room), dispatches the message to that person.

Coming back to `Person`'s implementations of `Say()` and `PrivateMessage()`, here they are:

```
public void Say(string message) => Room.Broadcast(Name, message);

public void PrivateMessage(string who, string message)
{
  Room.Message(Name, who, message);
}
```

As for `Receive()`, well, this is a good place to actually display the message on-screen as well as add it to the chat log.

```
public void Receive(string sender, string message)
{
  string s = $"{sender}: '{message}'";
  WriteLine($"[{Name}'s chat session] {s}");
  chatLog.Add(s);
}
```

We go the extra mile here by displaying not just who the message came from, but whose chat session we're currently in – this will be useful for diagnosing who said what and when.

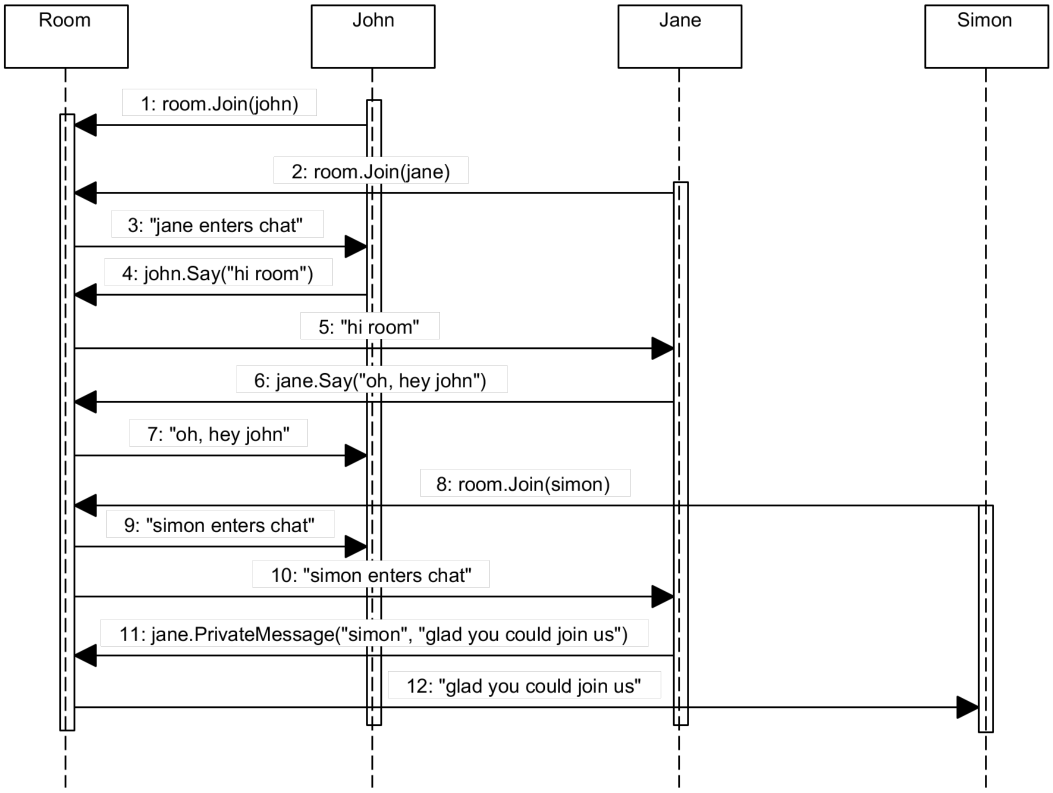
Here's the scenario that we'll run through:

```
var room = new ChatRoom();

var john = new Person("John");
var jane = new Person("Jane");

room.Join(john);
room.Join(jane);

john.Say("hi room");
jane.Say("oh, hey john");

var simon = new Person("Simon");
room.Join(simon);
simon.Say("hi everyone!");

jane.PrivateMessage("Simon", "glad you could join us!");
```

Here is the output:

```
[john's chat session] room: "jane joins the chat"
[jane's chat session] john: "hi room"
[john's chat session] jane: "oh, hey john"
[john's chat session] room: "simon joins the chat"
[jane's chat session] room: "simon joins the chat"
[john's chat session] simon: "hi everyone!"
[jane's chat session] simon: "hi everyone!"
[simon's chat session] jane: "glad you could join us, simon"
```

And here's an illustration of the chat room operations:



Mediator with Events

In the chat room example, we've encountered a consistent theme: the participants need notification whenever someone posts a message. This seems like a perfect scenario for the Observer pattern, which is discussed later in the book: the idea of the mediator having an event that is shared by all participants; participants can then subscribe to the event to receive notifications, and they can also cause the event to fire, thus triggering said notifications.

Instead of redoing the chat room once again, let's go for a simpler example: imagine a game of football (soccer for my readers in USA) with players and a football coach. When the coach sees their team scoring, they naturally want to congratulate the player. Of course, they need some information about the event, like *who* scored the goal and how many goals they have scored so far.

We can introduce a base class for any sort of event data:

```
abstract class GameEventArgs : EventArgs
{
  public abstract void Print();
}
```

I've added the `Print()` deliberately to print the event's contents to the command line. Now, we can derive from this class in order to store some goal-related data:

```
class PlayerScoredEventArgs : GameEventArgs
{
  public string PlayerName;
  public int GoalsScoredSoFar;

  public PlayerScoredEventArgs
    (string playerName, int goalsScoredSoFar)
  {
    PlayerName = playerName;
    GoalsScoredSoFar = goalsScoredSoFar;
  }

  public override void Print()
  {
    WriteLine($"{PlayerName} has scored! " +
              $"(their {GoalsScoredSoFar} goal)");
  }
}
```

We are once again going to build a mediator, but it will have *no* behaviors! Seriously, with an event-driven infrastructure, they are no longer needed:

```
class Game
{
  public event EventHandler<GameEventArgs> Events;

  public void Fire(GameEventArgs args)
  {
    Events?.Invoke(this, args);
  }
}
```

As you can see, we've just made a central place where all game events are being generated. The generation itself is polymorphic: the event uses a `GameEventArgs` type, and you can test the argument against the various types available in your application. The `Fire()` utility method just helps us safely raise the event.

We can now construct the `Player` class. A player has a name, the number of goals they scored during the match, and a reference to the mediator `Game`, of course:

```
class Player
{
  private string name;
  private int goalsScored = 0;
  private Game game;

  public Player(Game game, string name)
  {
    this.name = name;
    this.game = game;
  }

  public void Score()
  {
    goalsScored++;
    var args = new PlayerScoredEventArgs(name, goalsScored);
    game.Fire(args);
  }
}
```

The `Player.Score()` method is where we make `PlayerScoredEventArgs` and post them for all subscribers to see. Who gets this event? Why, a `Coach`, of course:

```
class Coach
{
  private Game game;

  public Coach(Game game)
  {
    this.game = game;

    // celebrate if player has scored <3 goals
    game.Events += (sender, args) =>
    {
      if (args is PlayerScoredEventArgs scored
          && scored.GoalsScoredSoFar < 3)
      {
        WriteLine($"coach says: well done, {scored.PlayerName}");
      }
    };
  }
}
```

The implementation of the `Coach` class is trivial; our coach doesn't even get a name. But we do give him a constructor where a subscription is created to game's `Events` such that, whenever something happens, the coach gets to process the event data in the provided lambda.

Notice that the argument type of the lambda is `GameEventArgs` – we don't know if a player has scored or has been sent off, so we need a cast to determine we've got the right type.

The interesting thing is that all the magic happens at the set-up stage: there's no need to explicitly subscribe to particular events. The client is free to create objects using their constructors and then, when the player scores, the notifications are sent:

```
var game = new Game();
var player = new Player(game, "Sam");
var coach = new Coach(game);

player.Score(); // coach says: well done, Sam
player.Score(); // coach says: well done, Sam
player.Score(); //
```

The output is only 2 lines long because, on the third goal, the coach isn't impressed anymore.

Introduction to MediatR

MediatR is one of a number of libraries written to provide a shrink-wrapped Mediator implementation in .NET[36]. It provides the client a central `Mediator` component, as well as interfaces for requests and request handlers. It supports both synchronous and async/await paradigms, and provides support for both directed messages as well as broadcasting.

As you may have guessed, MediatR is designed to work with an IoC container. It comes with examples for how to get it running with most popular containers out there; I'll be using Autofac for my examples.

The first steps are general: we simply set up MediatR under our IoC container, and also register our own types through the interfaces they implement.

```
var builder = new ContainerBuilder();
builder.RegisterType<Mediator>()
  .As<IMediator>()
  .InstancePerLifetimeScope(); // singleton

builder.Register<ServiceFactory>(context =>
{
  var c = context.Resolve<IComponentContext>();
  return t => c.Resolve(t);
});
```

[36]MediatR is available on NuGet; source code can be found at https://github.com/jbogard/MediatR.

```
builder.RegisterAssemblyTypes(typeof(Demo).Assembly)
  .AsImplementedInterfaces();
```

The central `Mediator`, which we registered as a singleton, is in charge of routing requests to request handlers and getting responses from them. Each request is expected to implement the `IRequest<T>` interface, where `T` is the type of the response that is expected for this request. If there is no data to return, you can use a non-generic `IRequest` instead.

Here's a simple example:

```
public class PingCommand : IRequest<PongResponse> {}
```

So in our trivial demo, we intend to send a `PingCommand` and receive a `PongResponse`. The response doesn't have to implement any interface; we'll define it like this:

```
public class PongResponse
{
  public DateTime Timestamp;

  public PongResponse(DateTime timestamp)
  {
    Timestamp = timestamp;
  }
}
```

The glue that connects requests and responses together is MediatR's `IRequestHandler` interface. It has a single member called `Handle` that takes a request and a cancellation token and returns the result of the call:

```
[UsedImplicitly]
public class PingCommandHandler
  : IRequestHandler<PingCommand, PongResponse>
{
  public async Task<PongResponse> Handle(PingCommand request,
    CancellationToken cancellationToken)
  {
    return await Task
      .FromResult(new PongResponse(DateTime.UtcNow))
      .ConfigureAwait(false);
  }
}
```

Note the use of the async/await paragigm above, with the `Handle` method returning a `Task<T>`. If you don't actually need your request to produce a response, then instead of using an `IRequestHandler` you can use the `AsyncRequestHandler` base class, whose `Handle()` method returns a humble non-generic `Task`. Oh, and in case your request is synchronous, you can inherit from `RequestHandler<TRequest, TResponse>` class instead.

This is all that you need to do to actually set up two components and get them talking through the central mediator. Note that the mediator itself does not feature in any of the classes we've created: it works behind the scenes.

Putting everything together, we can use our set-up as follows:

```
var container = builder.Build();
var mediator = container.Resolve<IMediator>();
var response = await mediator.Send(new PingCommand());
Console.WriteLine($"We got a pong at {response.Timestamp}");
```

You'll notice that request/response messages are targeted: they are dispatched to a single handler. MediatR also supports notification messages, which can be dispatched to multiple handlers. In this case, your request needs to implement the `INotification` interface:

```
public class Ping : INotification {}
```

And now you can create any number of `INotification<Ping>` classes that get to process these notifications:

```
public class Pong : INotificationHandler<Ping>
{
    public Task Handle(Ping notification, CancellationToken cancellationToken)
    {
        Console.WriteLine("Got a ping");
        return Task.CompletedTask;
    }
}
public class AlsoPong : INotificationHandler<Ping> { ... }
```

For notifications, instead os using the `Send()` method, we use the `Publish()` method:

```
await mediator.Publish(new Ping());
```

There is more information about MediatR available on its official Wiki page (https://github.com/jbogard/MediatR/wiki).

Summary

The Mediator design pattern is all about having an in-between component that everyone in a system has a reference to and can use to communicate with one another. Instead of direct references, communication can happen through identifiers (usernames, unique IDs, GUIDs, etc).

The simplest implementation of a mediator is a member list and a function that goes through the list and does what it's intended to do – whether on every element of the list, or selectively.

A more sophisticated implementation of Mediator can use events to allow participants to subscribe (and unsubscribe) to things happening in the system. This way, messages sent from one component to another can be treated as events. In this set-up, it is also easy for participants to unsubscribe to certain events if they are no longer interested in them or if they are about to leave the system altogether.

Memento

When we looked at the Command design pattern, we noted that recording a list of every single change theoretically allows you to roll back the system to any point in time – after all, you've kept a record of all the modifications.

Sometimes, though, you don't really care about playing back the state of the system, but you *do* care about being able to roll back the system to a *particular* state, if need be.

This is precisely what the Memento pattern does: it typically stores the state of the system and returns it as a dedicated, read-only object with no behavior of its own. This 'token', if you will, can be used only for feeding it back into the system to restore it to the state it represents.

Let's look at an example.

Bank Account

Let's use an example of a bank account that we've made before…

```
public class BankAccount
{
  private int balance;

  public BankAccount(int balance)
  {
    this.balance = balance;
  }

  // todo: everything else :)
}
```

…but now we decide to make a bank account with a `Deposit()`. Instead of it being `void` as in previous examples, `Deposit()` will now be made to return a `Memento`:

```
public Memento Deposit(int amount)
{
  balance += amount;
  return new Memento(balance);
}
```

and the Memento will then be usable for rolling back the account to the previous state:

```
public void Restore(Memento m)
{
  balance = m.Balance;
}
```

As for the memento itself, we can go for a trivial implementation:

```
public class Memento
{
  public int Balance { get; }

  public Memento(int balance)
  {
    Balance = balance;
  }
}
```

You'll notice that the `Memento` class is immutable. Imagine if you *could*, in fact, change the balance: you could roll back the account to a state it was never in!

And here is how one would go about using such a set-up:

```
var ba = new BankAccount(100);
var m1 = ba.Deposit(50);
var m2 = ba.Deposit(25);
WriteLine(ba); // 175

// restore to m1
ba.Restore(m1);
WriteLine(ba); // 150

// restore back to m2
ba.Restore(m2);
WriteLine(ba); // 175
```

This implementation is good enough, through there are some things missing. For example, you never get a Memento representing the opening balance because a constructor cannot return a value. You could add an out parameter, of course, but that's just too ugly.

Undo and Redo

What if you were to store *every* Memento generated by BankAccount? In this case, you'd have a situation similar to our implementation of the Command pattern, where undo and redo operations are a byproduct of this recording. Let's see how we can get undo/redo functionality with a Memento.

We'll introduce a new BankAccount class that's going to keep hold of every single Memento it ever generates:

```
public class BankAccount
{
  private int balance;
  private List<Memento> changes = new List<Memento>();
  private int current;

  public BankAccount(int balance)
  {
    this.balance = balance;
    changes.Add(new Memento(balance));
  }
}
```

We have now solved the problem of returning to the initial balance: the memento for the initial change is stored as well. Of course, this memento isn't actually returned, so in order to roll back to it, well, I suppose you could implement some `Reset()` function or something – totally up to you.

The `BankAccount` class has a `current` member that stores the index of the latest momento. Hold on, why do we need this? Isn't it the case that `current` will always be one less than the list of `changes`? Only if you want to support undo/rollback operations; if you want redo operations too, you need this!

Now, here's the implementation of the `Deposit()` method:

```
public Memento Deposit(int amount)
{
  balance += amount;
  var m = new Memento(balance);
  changes.Add(m);
  ++current;
  return m;
}
```

There are several things that happen here:

- The balance is increased by the amount you wanted to deposit.
- A new memento is constructed with the new balance and added to the list of changes.
- We increase the `current` value (you can think of it as a pointer into the list of `changes`).

Now here comes the fun stuff. We add a method to restore the account state based on a memento:

```
public void Restore(Memento m)
{
  if (m != null)
  {
    balance = m.Balance;
    changes.Add(m);
    current = changes.Count - 1;
  }
}
```

The restoration process is significantly different to the one we've looked at earlier. First, we actually check that the memento is initialized – this is relevant because we now have a way of signaling no-ops: just return a default value. Also, when we restore a memento, we actually add that memento to the list of changes so an undo operation will work correctly on it.

Now, here is the (rather tricky) implementation of `Undo()`:

```
public Memento Undo()
{
  if (current > 0)
  {
    var m = changes[--current];
    balance = m.Balance;
    return m;
  }
  return null;
}
```

We can only `Undo()` if `current` points to a change that is greater than zero. If that's the case, we move the pointer back, grab the change at that position, apply it and then return that change. If we cannot roll back to a previous memento, we return `null`, which should explain why we check for `null` in `Restore()`.

The implementation of `Redo()` is very similar:

```
public Memento Redo()
{
  if (current + 1 < changes.Count)
  {
    var m = changes[++current];
    balance = m.Balance;
    return m;
  }
  return null;
}
```

Again, we need to be able to redo something: if we can, we do it safely, if not – we do nothing and return `null`. Putting it all together, we can now start using the undo/redo functionality:

```
var ba = new BankAccount(100);
ba.Deposit(50);
ba.Deposit(25);
WriteLine(ba);

ba.Undo();
WriteLine($"Undo 1: {ba}"); // Undo 1: 150

ba.Undo();
WriteLine($"Undo 2: {ba}"); // Undo 2: 100

ba.Redo();
WriteLine($"Redo 2: {ba}"); // Redo 2: 150
```

Using Memento for Interop

Sometimes, managed code is not enough. For example, you need to run some calculations on the GPU and those are (typically) programmed using CUDA C and the like. You end up having to use a C or C++ library from your C# code, so you make calls from the managed (.NET) side to the unmanaged (native code) side.

This isn’t really a problem if you want to pass simple bits of data, such as numbers or arrays, back and forth. .NET has functionality for pinning an array and sending it to the ‘unmanaged’ side for processing. It works fine, most of the time.

The problems arise when you allocate some object-oriented construct (i.e., a class) inside unmanaged code and want to return that to the managed caller. Nowadays, this is typically handled by serializing (encoding) all the data on one side and then unpacking it on the other side. There are plenty of approaches here, including simple ones such as returning XML or JSON or complicated, industry-grade solutions such as Google’s Protocol Buffers.

In some cases, though, you don’t really need to return the full object itself. Instead, you simply want to return a handle so that this handle can be subsequently used on the unmanaged side again. You don’t even need the extra memory traffic passing objects back and forth. There are many reasons why you’d want to do this, but the main reason is that you want only one side to manage the object’s lifetime, since managing it on both sides is a nightmare that nobody really needs.

What you do in this case is you return a Memento. This can be anything – a string identifier, an integer, a GUID – anything that lets you refer to the object later on. The managed side then holds on to the token and uses that token to pass back to the unmanaged side when some operations on the underlying object are required.

This approach introduces an issue with lifetime management. Suppose we want the underlying object to live for as long as we have the token. How can we implement this? Well, this would mean that, on the unmanaged side, the token lives forever, whereas on the managed side, we wrap it in an `IDisposable` with the `Dispose()` method sending a message back to the unmanaged side that the token has been disposed. But what if we copy the token and have two or more instances of it? Then we end up having to build a reference-counted system for tokens: something that is quite possible, but introduces extra complexity in our system.

There is also a symmetric problem: what if the managed side has destroyed the object that the token represents? If we try to use the token, additional checks need to be made to ensure the token is actually valid, and some sort of meaningful return value needs to be given to the unmanaged call in order to tell the managed side that the token has gone stale. Again, this is extra work.

Summary

The Memento pattern is all about handing out tokens that can be used to restore the system to a prior state. Typically, the token contains all the information necessary to move the system to a particular state, and, if it's small enough, you can also use it to record *all* the states of the system so as to allow not just the arbitrary resetting of the system to a prior state, but controlled navigation backwards (undo) and forwards (redo) of all the states the system was in.

One design decision that I made in the demos above is to make the memento a `class`. This allows me to use the `null` value to encode the absence of a memento to operate opon. If we wanted to make it a `struct` instead, we would have to redesign the API so that, instead of `null`, the `Restore()` method would be able to take either a `Nullable<Memento>`, some `Option<Memento>` type (.NET doesn't have a built-in option type yet) or a memento possessing some easily identifiable trait (e.g., a balance of `int.MinValue`).

Null Object

We don't always choose the interfaces we work with. For example, I'd rather have my car drive me to my destination by itself, without me having to give 100% of my attention to the road and the dangerous lunatics driving next to me. And it's the same with software: sometimes you don't really want a piece of functionality, but it's built into the interface. So what do you do? You make a Null Object.

Scenario

Suppose you inherited a library that uses the following interface:

```
public interface ILog
{
  void Info(string msg);
  void Warn(string msg);
}
```

The library uses this interface to operate on bank accounts such as:

```
public class BankAccount
{
  private ILog log;
  private int balance;

  public BankAccount(ILog log)
  {
    this.log = log;
  }

  // more members here
}
```

In fact, `BankAccount` can have methods similar to:

```
public void Deposit(int amount)
{
  balance += amount;
  log.Info($"Deposited ${amount}, balance is now {balance}");
}
```

So, what's the problem here? Well, if you *do* need logging, there's no problem, you just implement your own logging class…

```
class ConsoleLog : ILog
{
  public void Info(string msg)
  {
    WriteLine(msg);
  }

  public void Warn(string msg)
  {
    WriteLine("WARNING: " + msg);
  }
}
```

…and you can use it straight away. But what if you *don't want logging at all*?

Intrusive Approaches

If you are prepared to break the Open-Closed Principle, there are a couple of intrusive approaches (with varying degrees of intrusiveness) that help you navigate around this situation.

The simplest approach, and also the ugliest, is to change the interface to an abstract class, i.e., change `ILog` to

```
public abstract class ILog
{
  void Info(string msg) {}
  void Warn(string msg) {}
}
```

You might want to follow up this change with a Rename refactoring from `ILog` to `Log`, but hopefully, the approach is obvious: by providing default no-op implementations in the base class, you can now simply make a dummy inheritor of this new `ILog` and supply it to whoever needs it. Or you can go further, make it non-abstract, and then `ILog` *is* your Null Object insofar as no-op behavior is concerned.

This approach can easily break things â€” after all, you might have clients who are explicitly assuming that ILog is an interface, so they could be implementing it *together with other interfaces* in their classes, which means this modification will break existing code.

Another alternative to the above is to simply add `null` checks everywhere. You could then rewrite the `BankAccount` constructor to have a default null argument:

```
public BankAcccount(ILog log = null) { ... }
```

With this change, you now need to change every single call on the log to a safe call, e.g., `log?.Info(...)`. This will work, but it can result in a huge number of changes if the log is used all over the place. There's also a small issue with the fact that using `null` for absence is not idiomatically correct (not obvious) â€” perhaps a better approach would be to use some `Option<T>` type, but such use would result in even more drastic changes accross the codebase.

Null Object Virtual Proxy

The final intrusive approach requires just one change inside the `BankAccount` class, and is the least harmful: it involves the construction of a Virtual Proxy (see the Proxy chapter) over an `ILog`. Essentially, we make a proxy/decorator over a log where the underlying is allowed to be `null`:

```
class OptionalLog: ILog
{
  private ILog impl;

  public OptionalLog(ILog impl) { this.impl = impl; }

  public void Info(string msg) { impl?.Info(msg); }

  public void Warn(string msg) { impl?.Warn(msg); }
}
```

Then, we change the `BankAccount` constructor, adding both an optional `null` value as well as the use of the wrapper in the body. In fact, if you can bear just one more line in the `BankAccount` class, we can do a neat trick by introducing a nice, descriptive constant called `NoLogging` and using it instead:

```
private const ILog NoLogging = null;

public BankAccount([CanBeNull] ILog log = NoLogging)
{
  this.log = new OptionalLog(log);
}
```

This approach is probably the least intrusive and most hygienic, allowing a `null` where such a value was hitherto not allowed while, at the same time, using the name of the default value to hint at what's going on.

Null Object

There are situations where none of the intrusive approaches will work, the most obvious being the case where you don't actually own the code that is using the relevant component. In this case, we need to construct a separate Null Object, which gives rise to the pattern we are discussing.

Look at `BankAccount`'s constructor once again:

```
public BankAccount(ILog log)
{
  this.log = log;
}
```

Since the constructor takes a logger, it is *unsafe* to assume that you can get away with just passing it a `null`. `BankAccount` *could* be checking the reference internally before dispatching on it, but you don't know that it does, and without extra documentation it's impossible to tell.

As a consequence, the only thing that would be reasonable to pass into `BankAccount` is a *null object* – a class which conforms to the interface but contains no functionality:

```
public sealed class NullLog : ILog
{
  public void Info(string msg) { }
  public void Warn(string msg) { }
}
```

Notice that the class is `sealed`: this is a design choice that presupposes that there is no point in inheriting from an object that deliberately has no behavior. Essentially, `NullLog` is a worthless parent.

Dynamic Null Object

In order to construct a correct Null Object, you have to implement every member of the required interface. Booo-ring! Can't we just write a single method that says "please just do nothing on *any* call"? Turns out we can, thanks to the DLR.

For this example, we are going to make a type called `Null<T>` that will inherit from `DynamicObject` and simply provide a no-op response to any method that's called on it:

```
public class Null<T> : DynamicObject where T:class
{
  public override bool TryInvokeMember(InvokeMemberBinder binder,
    object[] args, out object result)
  {
    var name = binder.Name;
    result = Activator.CreateInstance(binder.ReturnType);
    return true;
  }
}
```

As you can see, all this dynamic object does is construct a default instance of whatever type the method in question actually returns. So if our logger returned an `int` indicating the number of lines writte to the log, our dynamic object would just return `0` (zero).

Now, I have neglected to mention what the `T` in `Null<T>` actually is. As you may have guessed, that's the interface that we need a no-op object for. We can create a utility property getter to actually construct instances of `Null<T>` that satisfy the interface `T`. For this, we are going to use the ImpromptuInterface library.[37]

```
public static T Instance
{
  get
  {
    if (!typeof(T).IsInterface)
      throw new ArgumentException("I must be an interface type");

    return new Null<T>().ActLike<T>();
  }
}
```

In the above, the `ActLike()` method from ImpromptuInterface takes a `dynamic` object and conforms it at runtime to the required interface `T`.

Putting everything together, we can now write the following:

[37] ImpromptuInterface is an open-source dynamic 'duck casting' library built on top of DLR and `Reflection.Emit`. Its source code is available at https://github.com/ekonbenefits/impromptu-interface and you can install it directly from NuGet.

```
var log = Null<ILog>.Instance;
var ba = new BankAccount(log);
ba.Deposit(100);
ba.Withdraw(200);
```

Once again, this code has a computational cost related to the construction of a dynamic object that not only does no-ops, but also conforms to the chosen interface.

Summary

The Null Object pattern raises an issue of API design: what kinds of assumptions can we make about the objects we depend upon? If we are taking a reference, do we then have an obligation to check this reference on every use?

If you feel no such obligation, then the only way the client can implement a Null Object is to contruct a no-op implementation of the required interface and pass that instance in. That said, this only works well with methods: if the object's fields are also being used, for example, then you are in real trouble. Same goes for non-void methods where the return values are actually used for something.

If you want to proactively support the idea of Null Objects being passed as arguments, you need to be explicit about it: either specify the parameter type as some `Optional`, give the parameter a default value that hints at a possible `null` or just write documentation that explains what kind of value is expected at this location.

Observer

The Observer pattern, quite simply, lets one component notify other components that something happened. The pattern is used all over the place: for example, when binding data to UI, we can program domain objects such that, when they change, they generate notifications that the UI can subscribe to and update the visuals.

Events

The Observer pattern is a popular and necessary pattern, so it is not surprising that the designers of C# decided to incorporate the pattern into the language wholesale with the use of the `event` keyword. The use of events in C# typically uses a convention that mandates the following:

- Events can be members of a class, and are decorated with the `event` keyword.
- Event handlers – methods that are called whenever an event is raised – are attached to the event with the `+=` operator and are detached with the `-=` operator.
- An event handler typically takes two arguments:
  - An `object` reference to who exactly fired the event.
  - An object that (typically) derives from `EventArgs` that contains any necessary information about the event.

The exact *type* of an event that is used is typically a delegate. Just like the `Action`/`Func` wrappers for lambdas, the delegate wrappers for events are called `EventHandler` and exist in both non-generic (that takes an `EventArgs`) and a generic (that takes a type parameter that derives from `EventArgs`) second argument. The first argument is always an `object`.

Here's a trivial example: suppose, whenever a person falls ill, we call a doctor. First of all we define event arguments; in our case we just need the address to send the doctor to:

```
public class FallsIllEventArgs : EventArgs
{
  public string Address;
}
```

Now, we can implement a `Person` type, which can look like this:

```
public class Person
{
  public void CatchACold()
  {
    FallsIll?.Invoke(this,
      new FallsIllEventArgs { Address = "123 London Road" });
  }

  public event EventHandler<FallsIllEventArgs> FallsIll;
}
```

As you can see, we are using a strongly typed `EventHandler` delegate to expose a public event. The `CatchACold()` method is used to raise the event, with the safe access `?.` operator being used to ensure that, if the event doesn't have any subscribers, we don't get a `NullReferenceException`.

All that remains is to set up a scenario and provide an event handler:

```
static void Main()
{
  var person = new Person();
  person.FallsIll += CallDoctor;
  person.CatchACold();
}

private static void CallDoctor(object sender, FallsIllEventArgs eventArgs)
{
  Console.WriteLine($"A doctor has been called to {eventArgs.Address}");
}
```

The event handler can be an ordinary (member) method, a local function or a lambda – your choice. The signature is mandated by the original delegate; since we're using a strongly typed `EventHandler` variant, the second argument is `FallsIllEventArgs`. As soon as `CatchACold()` is called, the `CallDictor()` method is triggered.

Any given event can have more than one handler (C# delegates are multicast, after all). Removal of event handlers is typically done with the `-=` operator. When all subscribers have unsubscribed from an event, the event instance is set to `null`.

Weak Event Pattern

Did you know that .NET programs can have memory leaks? Not in the C++ sense, of course, but it *is* possible to keep holding on to an object for longer than necessary. Specifically, you can make an object and set its reference to `null` but it will still be alive. How? Let me show you.

First, let's make a `Button` class:

```
public class Button
{
  public event EventHandler Clicked;

  public void Fire()
  {
    Clicked?.Invoke(this, EventArgs.Empty);
  }
}
```

So now let's suppose we have this button in a window. For the sake of simplicity, I'll just stick it into a `Window` constructor:

```
public class Window
{
  public Window(Button button)
  {
    button.Clicked += ButtonOnClicked;
  }

  private void ButtonOnClicked(object sender, EventArgs eventArgs)
  {
    WriteLine("Button clicked (Window handler)");
  }

  ~Window()
  {
    WriteLine("Window finalized");
  }
}
```

Looks innocent enough, except it's not. If you make a button and a window, then set the window to `null`, it will still be alive! Proof:

```
var btn = new Button();
var window = new Window(btn);
var windowRef = new WeakReference(window);
btn.Fire();

window = null;

FireGC();
WriteLine($"Is window alive after GC? {windowRef.IsAlive}"); // True
```

The reason why the window reference is still alive is that it has a subscription to the button. When a button is clicked, the expection is that something sensible happens: since there is a subscription to this event, the object that happens to have made this subscription cannot be allowed to day, even if the only reference to that object has been set to `null`. This is a memory leak in the .NET sense.

How can we fix this? One approach would be to use the `WeakEventManager` class from `System.Windows`. This class is specifically designed to allow the listener's handlers to be garbage-collected even if the source object persists. This class is very simple to use:

```
public class Window2
{
  public Window2(Button button)
  {
    WeakEventManager<Button, EventArgs>
      .AddHandler(button, "Clicked", ButtonOnClicked);
  }
  // rest of class same as before
}
```

Repeating the scenario again, this `Window2` implementation gives a `windowRef.IsAlive` result of `False`, as desired.

Event Streams

With all these discussions of Observer, you might be interested to learn that the .NET framework comes with two interfaces: `IObserver<T>` and `IObservable<T>`. These interfaces, which were coincidental with the release of Reactive Extensions (Rx), are meant primarily to deal with reactive streams. While it is not my intention to discuss the entirety of Reactive Extensions, these two interfaces are worth mentioning.

Let's start with `IObservable<T>`. This is an interface that is generally similar to the interface of a typical .NET event. The only difference is that, instead of using the += operator for subscription, this interface requires that you implement a method called `Subscribe()`. This method takes an `IObserver<T>` as its only parameter. Remember, this is an interface, and, unlike in the case of events/delegates, there is no prescribed storage mechanism. You are free to use anything you want.

There is some extra icing on the cake: the notion of *un*subscription is explicitly supported in the interface. The `Subscribe()` method returns an `IDisposable` with the understanding that the return token (Memento pattern at work!) has a `Dispose()` method that unsubscribes the observer from the observable.

The second piece of the puzzle is the `IObserver<T>` interface. It is designed to provide push-based notifications through three specific methods:

- `OnNext(T)` gets invoked whenever a new event occurs.
- `OnCompleted()` gets invoked when the source has no more data to give.
- `OnError()` gets invoked whenever the observer has experienced an error condition.

Once again, this is just an interface, and how you handle this is up to you. For example, you can completely ignore both `OnCompleted()` and `OnError()`.

So, given these two interfaces, the implementation our trivial doctor-patient example is suddenly a lot less trivial. First of all, we need to encapsulate the idea of an *event subscription*. The reason why this is required is because we need a memento that implements `IDisposable` through which unsubscription can happen.

```
private class Subscription : IDisposable
{
  private Person person;
  public IObserver<Event> Observer;

  public Subscription(Person person, IObserver<Event> observer)
  {
    this.person = person;
    Observer = observer;
  }

  public void Dispose()
  {
    person.subscriptions.Remove(this);
  }
}
```

This class is an inner class of `Person`, which is a good hint at the growing complexity of any object that wants to support event streams. Now, coming back to `Person`, we want it to implement the `IObservable<T>` interface. But what is T? Unlike the conventional events, there are no guidelines mandating that we inherit from EventArgs – sure, we could continue using that type[38], or we could construct our own, completely arbitrary, hierarchy:

```
public class Event
{
  // anything could be here
}

public class FallsIllEvent : Event
{
  public string Address;
}
```

Moving on, we now have a base class Event, so we can declare Person to be a generator of such events. As a consequence, our Person type would implement IObservable<Event> and would take an IObserver<Event> in its Subscribe() method. Here is the entire Person class with the body of the Subscription inner class omitted:

```
public class Person : IObservable<Event>
{
  private readonly HashSet<Subscription> subscriptions
    = new HashSet<Subscription>();

  public IDisposable Subscribe(IObserver<Event> observer)
  {
    var subscription = new Subscription(this, observer);
    subscriptions.Add(subscription);
    return subscription;
  }

  public void CatchACold()
  {
    foreach (var sub in subscriptions)
      sub.Observer.OnNext(new FallsIllEvent {Address = "123 London Road"});
```

[38]By the way, System.EventArgs is an empty type. All it has is a default constructor (empty) and a static member EventArgs.Empty that is a Singleton Null Object (double pattern headshot!) that indicates the event arguments have no data.

```
  }

  private class Subscription : IDisposable { ... }
}
```

I'm sure you'll agree that this is a lot more complicated than just publishing a single `event` for clients to subscribe to! But there are advantages to this: for example, you can choose your own policy with respect to repeat subscriptions, i.e., situations when a subscriber is trying to subscribe to some event *again*. One thing worth noting is that `HashSet<Subscription>` is not a thread-safe container. This means that if you want `Subscribe()` and `CatchACold()` to be callable concurrently, you would need to either use a thread-safe collection, locking or perhaps something even fancier, like an `ImmutableList`.

The problems don't end there. Remember, a subscriber has to implement an `IObserver<Event>` now. This means that, to support the scenario we've had previously shown, we would have to write the following:

```
public class Demo : IObserver<Event>
{
  static void Main(string[] args)
  {
    new Demo();
  }

  public Demo()
  {
    var person = new Person();
    var sub = person.Subscribe(this);
  }

  public void OnNext(Event value)
  {
    if (value is FallsIllEvent args)
      WriteLine($"A doctor has been called to {args.Address}");
  }

  public void OnError(Exception error){}
  public void OnCompleted(){}
}
```

This is, once again, quite a mouthful. We could have simplified the subscription by using a special `Observable.Subscribe()` static method, but `Observable` (without the `I`) is part of Reactive Extensions, a separate library that you may or may not want to use.

So this is how you can build an Observer pattern using .NET's own interfaces, without using the `event` keyword. The main advantage of this approach is that the stream of events that is generated by an `IObservable` can be directly fed into various Rx operators. For example, using `System.Reactive`, the entire demo program shown above cab turn into a single statement:

```
person
  .OfType<FallsIllEvent>()
  .Subscribe(args =>
    WriteLine($"A doctor has been called to {args.Address}"));
```

Property Observers

One of the most common Observer implementations in .NET is getting notifications when a property changes. This is necessary, for example, to update UI when the underlying data changes. This mechanism uses ordinary events as well as some interfaces that have become standard within .NET.

Property observers can get really complicated, so we'll cover them in steps, starting from basic interfaces and operations and moving onto the more complicated scenarios.

Basic Change Notification

The central piece of change notification in .NET is an interface called `INotifyPropertyChanged`:

```
public interface INotifyPropertyChanged
{
  /// <summary>Occurs when a property value changes.</summary>
  event PropertyChangedEventHandler PropertyChanged;
}
```

All this event does it expose an event that you're expected to use. Given a class `Person` having a property called Age, the typical implementation of this interface looks as follows:

```
public class Person : INotifyPropertyChanged
{
  private int age;

  public int Age
  {
    get => age;
    set
    {
      if (value == age) return;
      age = value;
      OnPropertyChanged();
    }
  }

  public event PropertyChangedEventHandler PropertyChanged;

  [NotifyPropertyChangedInvocator]
  protected virtual void OnPropertyChanged(
    [CallerMemberName] string propertyName = null)
  {
    PropertyChanged?.Invoke(this,
      new PropertyChangedEventArgs(propertyName));
  }
}
```

There is a lot to discuss here. First of all, the property gets a backing field. This is required in order to look at the previous value of the property before it is assigned.

Notice that the invocation of the `OnPropertyChanged()` method happens only if the property *did* change. If it didn't, there's no notification.

As far as the IDE-generated `OnPropertyChanged()` method is concerned, this method is designed to take in the name of the affected property via `[Caller-MemberName]` metadata and then, provided the `PropertyChanged` event has subscribers, notify those subscribers that the property with that name did, in fact, change.

You can, of course, build your own change notification mechanisms, but both WinForms and WPF are intrinsically aware of `INotifyPropertyChanged`, as are many other frameworks. So, if you need change notifications, I'd stick to this interface.

A special note needs to be added about `INotifyPropertyChanging` – an interface that is intended to send events indicating that a property is in the process of changing. This interface is very rarely used, if ever. It would have been nice to be able to use this property to *cancel* a property change, but sadly the interface makes no provisions for this. In actual fact, cancelation of property changes can be one of the reasons why you would want to implement *your own* interfaces instead of these.

Bidirectional Bindings

The `INotifyPropertyChanged` is very useful for notifying the user interface about the change of a property some label is bound to. But what if you have an edit box instead, and that edit box also needs to update the code element behind the scenes?

This is actually doable, and doesn't even result in infinite recursion! This problem generalizes to the following: how do you bind two properties such that changing the one changes the other, in other words, their values are always identical?

Let's try this. Suppose we have a `Product` that has a `Name` and we also have a `Window` that has a `ProductName`. We want `Name` and `ProductName` to be bound together.

```
var product = new Product{Name="Book"};
var window = new Window{ProductName = "Book"};

product.PropertyChanged += (sender, eventArgs) =>
{
  if (eventArgs.PropertyName == "Name")
  {
    Console.WriteLine("Name changed in Product");
    window.ProductName = product.Name;
  }
};

window.PropertyChanged += (sender, eventArgs) =>
{
  if (eventArgs.PropertyName == "ProductName")
  {
    Console.WriteLine("Name changed in Window");
    product.Name = window.ProductName;
  }
};
```

Common sense dictates that this code, when triggered, would cause a `StackOverflowException`: window affects product, product affects window, and so on. Except it doesn't happen. Why? Because the setter in both properties has a guard that checks that the value did, in fact, change. If it didn't, it does a `return` and no further notifications take place. So we're safe here.

The above solution works but frameworks such as WinForms try to shrink-wrap situations such as these into separate data binding objects. In a data binding, you specify the objects and their properties and how they tie together. Windows Forms, for example, uses property names (as strings), but nowadays we can be a little bit smarter and use expression trees instead.

So let's construct a `BidirectionalBinding` class that will, in its constructor, bind together two properties. For this, we need four pieces of information:

- The owner of the first property
- An expression tree accessing the first object's property
- The owner of the second property

- An expression tree accessing the second object's property

Sadly, it is impossible to reduce the number of parameters in this scenario, but at least they will be more or less human-readable. We'll also avoid using generics here, though they can, in theory, introduce additional type safety.

So, here is the entire class:

```
public sealed class BidirectionalBinding : IDisposable
{
  private bool disposed;

  public BidirectionalBinding(
    INotifyPropertyChanged first, Expression<Func<object>> firstProperty,
    INotifyPropertyChanged second, Expression<Func<object>> secondProperty)
  {
    if (firstProperty.Body is MemberExpression firstExpr
        && secondProperty.Body is MemberExpression secondExpr)
    {
      if (firstExpr.Member is PropertyInfo firstProp
          && secondExpr.Member is PropertyInfo secondProp)
      {
        first.PropertyChanged += (sender, args) =>
        {
          if (!disposed)
          {
            secondProp.SetValue(second, firstProp.GetValue(first));
          }
        };
        second.PropertyChanged += (sender, args) =>
        {
          if (!disposed)
          {
            firstProp.SetValue(first, secondProp.GetValue(second));
          }
        };
      }
    }
  }
  public void Dispose()
  {
    disposed = true;
  }
}
```

The above code depends on a number of preconditions regarding the expression trees, specifically:

- Each expression tree is expected to be a `MemberExpression`
- Each member expression is expected to access a property (thus, `PropertyInfo`)

If these conditions are met, we subscribe each property to each other's changes. There's an additional dispose guard added to this class to allow the user to stop processing the subscriptions if necessary.

The above is a trivial example of the kinds of things that can happen behind the scenes in frameworks that intend to intrinsically support data binding.

Property Dependencies

In Microsoft Excel, you can have cells contain calculations using values of other cells. This is very convenient: whenever a particular cell's value changes, Excel recalculates every single cell (including cells on other sheets) that this cell affected. And then *those* cells cause the recalculation of every cell dependent on *them*. And so it goes forever until the entire dependency graph is traversed, however long it takes. It's beautiful.

The problem with properties (and with the Observer pattern generally) is exactly the same: sometimes a part of a class not only generates notifications, but affects other parts of the class and then *those* members also generate their own event notifications. Unlike Excel, .NET doesn't have a built-in way of handling this, so such a situation can quickly turn into a real mess.

Let me illustrate. People aged 16 or older (could be different in your country) can vote, so suppose we want to be notified of changes to a person's voting rights:

```
public class Person : PropertyNotificationSupport
{
  private int age;

  public int Age
  {
    get => age;
    set
    {
      if (value == age) return;
      age = value;
      OnPropertyChanged();
    }
  }

  public bool CanVote => Age <= 16;
}
```

In the above, changes to a person's age should affect their ability to vote. However, we would also expect to generate appropriate change notifications for `CanVote`… but where? After all, `CanVote` has no setter!

You could try to put them into the Age setter, e.g.:

```
public int Age
{
  get => age;
  set
  {
    if (value == age) return;
    age = value;
    OnPropertyChanged();
    OnPropertyChanged(nameof(CanVote));
  }
}
```

This will work, but consider a scenario: what if age changes from 5 to 6? Sure, the age has changed, but `CanVote` has not, so why are we unconditionally doing a notification on it? This is incorrect. A functionally correct implementation would have to look something like the following:

```
set
{
  if (value == age) return;

  var oldCanVote = CanVote;

  age = value;
  OnPropertyChanged();

  if (oldCanVote != CanVote)
    OnPropertyChanged(nameof(CanVote));
}
```

As you can see, the only way to determine that `CanVote` has been affected is to cache its old value, perform the changes on `age`, then get its new value and check if it's been modified, and only then perform the notification.

Even without this particular pain point, the approach we've taken with property dependencies does not scale. In a complicated scenario where properties depend on other properties, how are we expected to track all the dependencies and make all the notifications? Clearly, some sort of centralized mechanism is needed to track all of this automatically.

Let's build such a mechanism. We'll construct a base class called `PropertyNotificationSupport` that will implement `INotifyPropertyChanged` and will also take care of dependencies. Here is its implementation:

```
public class PropertyNotificationSupport : INotifyPropertyChanged
{
  private readonly Dictionary<string, HashSet<string>> affectedBy
    = new Dictionary<string, HashSet<string>>();

  public event PropertyChangedEventHandler PropertyChanged;

  [NotifyPropertyChangedInvocator]
  protected virtual void OnPropertyChanged
    ([CallerMemberName] string propertyName = null)
  {
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    foreach (var affected in affectedBy.Keys)
      if (affectedBy[affected].Contains(propertyName))
        OnPropertyChanged(affected);
  }

  protected Func<T> property<T>(string name,
    Expression<Func<T>> expr) { ... }

  private class MemberAccessVisitor : ExpressionVisitor { ... }
}
```

This class is complicated, so let's go through this slowly and figure out what's going on here.

First, we have `affectedBy`, which is a dictionary that lists every property and a `HashSet` of properties affected by it. For example, if voting ability is affected by age and whether or not you're a citizen, this dictionary will contain a key of `"CanVote"` and values of `{"Age", "Citizen"}`.

We then modify the default `OnPropertyChanged()` implementation to ensure that the notifications happen both on the property itself, as well as all properties it affects. The only question now is – how do properties get enlisted in this dictionary?

It would be too much to ask the developers to populate this dictionary by hand. Instead, we do it automatically with the use of expression trees. A getter for a readonly property is provided to the base class as an expression tree, which completely changes the way dependent properties are constructed:

```
public class Person : PropertyNotificationSupport
{
  private readonly Func<bool> canVote;
  public bool CanVote => canVote();

  public Person()
  {
    canVote = property(nameof(CanVote),
      () => Citizen && Age >= 16);
  }

  // other members here
}
```

Clearly, everything has changed. The property is now initialized inside the constructor using the base class' `property()` method. That property takes an *expression tree*, parses it to find the dependency properties, then compiles the expression into an ordinary Func<T>:

```
protected Func<T> property<T>(string name, Expression<Func<T>> expr)
{
  Console.WriteLine($"Creating computed property for expression {expr}");

  var visitor = new MemberAccessVisitor(GetType());
  visitor.Visit(expr);

  if (visitor.PropertyNames.Any())
  {
    if (!affectedBy.ContainsKey(name))
      affectedBy.Add(name, new HashSet<string>());

    foreach (var propName in visitor.PropertyNames)
      if (propName != name)
        affectedBy[name].Add(propName);
  }
  return expr.Compile();
}
```

The parsing of the expression tree is done using a `MemberAccessVisitor`, a private, nested class that we've created. This class goes through the expression tree looking for member access and collects all the property names into a simple list:

```
private class MemberAccessVisitor : ExpressionVisitor
{
  private readonly Type declaringType;
  public readonly IList<string> PropertyNames = new List<string>();

  public MemberAccessVisitor(Type declaringType)
  {
    this.declaringType = declaringType;
  }

  public override Expression Visit(Expression expr)
  {
    if (expr != null && expr.NodeType == ExpressionType.MemberAccess)
    {
      var memberExpr = (MemberExpression)expr;
      if (memberExpr.Member.DeclaringType == declaringType)
      {
        PropertyNames.Add(memberExpr.Member.Name);
      }
    }

    return base.Visit(expr);
  }
}
```

Notice that we restrict ourselves to the declaring type of the owning class – handling a situation with property dependencies between classes is doable, but a lot more complicated.

Anyways, putting all of this together, we can now write something like the following:

```
var p = new Person();
p.PropertyChanged += (sender, eventArgs) =>
{
  Console.WriteLine($"{eventArgs.PropertyName} has changed");
};
p.Age = 16;
// Age has changed
// CanVote has changed
p.Citizen = true;
// Citizen has changed
// CanVote has changed
```

So it works. But our implementation is still far from ideal. If I were to change the age to 10 in the above, `CanVote` would still receive a notification, even though it shouldn't! That's because, at the moment, we're firing these notifications unconditionally. If we wanted to fire these only when the dependent properties have changed, we would have to resort to `INotifyPropertyChanging` (or a similar interface) where we would have to cache the old value of every affected property until the `INotifyPropertyChanged` call and then check that those have, in fact changed. I leave this as an exercise for the reader.

Finally, a small note. You can see some overcrowding happening inside property setters. 3 lines is already a lot, but if you factor in additional calls, such as, e.g., the use of `INotifyPropertyChanging`, then it makes sense to externalize the entire property setter. Turning each property into a `Property<T>` (see the Property Proxy section of the Proxy pattern) is a bit overkill, but we can imbue the base class with something like:

```
protected void setValue<T>(T value, ref T field,
  [CallerMemberName] string propertyName = null)
{
  if (value.Equals(field)) return;
  OnPropertyChanging(propertyName);
  field = value;
  OnPropertyChanged(propertyName);
}
```

With properties now simplifying to:

```
public int Age
{
  get => age;
  set => setValue(value, ref age);
}
```

Notice that, in the above, we must do `propertyName` propagation because the `[CallerMemberName]` attribute inside `OnPropertyChanged()` will no longer work for us out of the box.

Views

There's a big, huge, glaring problem with property observers: the approach is intrusive and clearly goes against the idea of Separation of Concerns. Change notification is a separate concern, so adding it right into your domain objects might not be the best idea.

Why not? Well, imagine you decide to change your mind and move from the use of INPC to the user of the `IObservable` interface. If you were to scatter INPC use throughout your domain objects, you'd have to meticulously go through each one, modifying each property to use the new paradigm, not to mention the fact that you'd have to modify those classes as well to stop using the old interfaces and start using the new ones. This is tedious and error-prone, and precisely the kind of thing we're trying to avoid by implementing better architecture.

So, if you want change notifications handled outside of the objects that change, where would you add them? It shouldn't be hard – after all, we've seen patterns such as Decorator that are designed for this exact purpose.

One approach is to put another object in front of your domain object that would handle change notifications and other things besides. This is what we would typically called a *view* – it is this thing that would be bound to UI, for example.

To use views, you would keep your objects simple, using ordinary properties (or even public fields!) without embellishing them with any extra behaviors:

```
public class Person
{
  public string Name;
}
```

In fact, it's worth keeping the data objects as simple as possible; this is what's known as a *data class* in languages such as Kotlin. Now what you do is build a view on top of the object. The view can incorporate other concerns, including property observers:

```
public class PersonView : View
{
  protected Person person;
  public PersonView(Person person)
  {
    this.person = person;
  }

  public string Name
  {
    get => person.Name;
    set {
      setValue(value, () => person.Name);
    }
  }
}
```

The above is, of course, a Decorator. It wraps the underlying object with mirroring getters/setters that perform the notifications. If you need even more complexity, this is the place to get it. For example, if you want property dependencies tracked in an expression tree, you'd do it in this constructor rather than in the constructor of the underlying object.

You'll notice that, in the above listing, I'm trying to be sly by hiding any implementation details. We simply inherit from some class `View` and we don't really care how it handles notifications: maybe it uses `INotifyPropertyChanged`, maybe it uses `IObservable`, maybe something else. We don't care.

The only real issue is how to call this class' setter considering that we want it to have information about both the name of the property we're assigning (just in case it's needed) and the value being assigned. There's no uniform solution to this problem and, obviously, the more information you pack into this improvised `setValue()` method, the better. If `person.Name` had been a field, things would be greatly simplified because we could simply pass a reference to that field to be assigned, but we'd still have to pass a `nameof()` for the base class to notify via INPC if necessary.

Observable Collections

If you bind a `List<T>` to a list box in either WinForms or WPF, changing the list won't update the UI. Why not? Because `List<T>` does not support the Observer pattern, of course – its individual members might, but the list as a whole has no explicit way of notifying that its contents have changed.. Admittedly, you could just make a wrapper where methods such as `Add()` and `Remove()` generate notifications. However, both WinForms and WPF come with *observable collections* – classes `BindingList<T>` and `ObservableCollection<T>` respectively.

Both of these types behave as a `Collection<T>`, but the operations generate additional notifications that can be used, for example, by UI components to update the presentation layer when the collections change. For example, `ObservableCollection<T>` implements the `INotifyCollectionChanged` interface which, in turn, has a `CollectionChanged` event. This event will tell you what action has been applied to the collection and will give you a list of both old and new items, as well as information about old and new starting indices: in other words, you get everything that you need in order to, say, redraw a list box correctly depending on the operation.

One important thing to note is that neither `BindingList<T>` nor `ObservableCollection<T>` are thread-safe. So if you plan to read/write these collections from multiple threads, you'd need to build a threading proxy (hey, Proxy pattern!). There are, in fact, two options here:

- Inherit from an observable collection and just put common collection operations such as `Add()` behind a lock; or
- Inherit from a concurrent collection (e.g., a `ConcurrentBag<T>`) and add `INotifyCollectionChanged` functionality.

You can find implementations of both of these approaches on StackOverflow and elsewhere. I prefer the first option as it's a lot simpler.

Observable LINQ

When we discussed property observers, we also managed to discuss the idea of properties that affect other properties. But that's not all they affect. For example, a property might be involved in a LINQ query which yields some result. So, how can we know that we need to re-query the data in a particular query when a property it depends on changes?

Over time, there had been frameworks such as CLinq (Continuous LINQ) and Bindable Linq that attempted to solve the problem of a LINQ query generiating the necessary events (i.e., `CollectionChanged`) when one of its constituent parts failed. Other frameworks exist, and I can offer no recommendations here about which one I would recommend you use. Keep in mind that these frameworks are attempting to solve a really difficult problem.

Declarative Subscriptions in Autofac

So far, most of our discussions have centered around the idea of *explicit*, imperative subscription to events, whether via the usual .NET mechanisms, Reactive Extension or something else. However, that's not the only way event subscriptions can happen.

You can also define event subscriptions *declaratively*. This is often made possible by the fact that applications make use of a central IoC container where the declarations can be found and event wireups can be made behind the scenes.

There are two popular approaches for declarative event wireups. The first uses attribute: you simply mark some method as `[Publishes("foo")]` and some other method, in some other class, as `[Subscribes("foo")]` and the IoC container makes a connection behind the scenes.

An alternative is to use interfaces, and that is what we are going to demonstrate, together with the use of the Autofac library. First, we define the notion of an event, and flesh out interfaces for the sending of the event and its handling:

```
public interface IEvent {}

public interface ISend<TEvent> where TEvent : IEvent
{
  event EventHandler<TEvent> Sender;
}

public interface IHandle<TEvent> where TEvent : IEvent
{
  void Handle(object sender, TEvent args);
}
```

We can now manufacture concrete implementations of events. For example, suppose we are handling click events, where a user can press a button a certain number of times (e.g., double-click it):

```
public class ButtonPressedEvent : IEvent
{
  public int NumberOfClicks;
}
```

We can now make a `Button` class that generates such events. For simplicity, we'll simply add a `Fire()` method that fires off the event. Adopting a declarative approach, we decorate the `Button` with the `ISend<ButtonPressedEvent>` interface:

```
public class Button : ISend<ButtonPressedEvent>
{
  public event EventHandler<ButtonPressedEvent> Sender;

  public void Fire(int clicks)
  {
    Sender?.Invoke(this, new ButtonPressedEvent
    {
      NumberOfClicks = clicks
    });
  }
}
```

And now, for the receiving side. Suppose we want to log button presses. This means we want to handle ButtonPressedEvents and, luckily, we already have an interface for this, too:

```
public class Logging : IHandle<ButtonPressedEvent>
{
  public void Handle(object sender, ButtonPressedEvent args)
  {
    Console.WriteLine(
      $"Button clicked {args.NumberOfClicks} times");
  }
}
```

Now, what we want, behind the scenes, is for our IoC container to automatically subscribe Logging to the Button.Sender event behind the scenes, without us having to do this manually. Let me first of all show you the monstrous piece of code that you would require to do this:

```
var cb = new ContainerBuilder();
var ass = Assembly.GetExecutingAssembly();

// register publish interfaces
cb.RegisterAssemblyTypes(ass)
  .AsClosedTypesOf(typeof(ISend<>))
  .SingleInstance();

// register subscribers
cb.RegisterAssemblyTypes(ass)
  .Where(t =>
    t.GetInterfaces()
      .Any(i =>
        i.IsGenericType &&
        i.GetGenericTypeDefinition() == typeof(IHandle<>)))
  .OnActivated(act =>
  {
    var instanceType = act.Instance.GetType();
    var interfaces = instanceType.GetInterfaces();
    foreach (var i in interfaces)
    {
      if (i.IsGenericType
          && i.GetGenericTypeDefinition() == typeof(IHandle<>))
      {
        var arg0 = i.GetGenericArguments()[0];
        var senderType = typeof(ISend<>).MakeGenericType(arg0);
        var allSenderTypes =
          typeof(IEnumerable<>).MakeGenericType(senderType);
        var allServices = act.Context.Resolve(allSenderTypes);
        foreach (var service in (IEnumerable) allServices)
        {
          var eventInfo = service.GetType().GetEvent("Sender");
          var handleMethod = instanceType.GetMethod("Handle");
          var handler = Delegate.CreateDelegate(
            eventInfo.EventHandlerType, null, handleMethod);
          eventInfo.AddEventHandler(service, handler);
        }
      }
    }
  })
  .SingleInstance()
  .AsSelf();
```

Let's go through what's happening in the above code step by step.

- First, we register all assembly types that implement `ISend<>`. There are no special steps that need to be taken there because they just need to exist somewhere. For the sake of simplicity, we register them as singletons – if that were not the case, the situation with the wireups would become even more comlpicated because the system would have to track each constructed instance[39].
- We then register the types that implement `IHandle<>`. This is where things get crazy because we specify an additional `OnActivated()` step that must be performed before an object is returned.

[39]From personal experience, there are many situations where you *do* need your IoC container to track all instances that have been constructed. One example is the Dynamic Prototyping approach (see the Dynamic Prototyping Bridge section of the Bridge chapter), where an in-process change of an object mandates an immediate replacement of all such objects that have been constructed during the application's lifetime.

- In this step, given this `IHandle<Foo>` type, we locate all types which implement the `ISend<Foo>` interface using Reflection. This is a rather tedious process.
- For each of the located types, we wire up the subscription. Again, this is done using reflection, and you can also see some magic strings here and there.

With this set-up in place, we can build the container, resolve both a `Button` and a `Logging` component, and the subscriptions will be done behind the scenes:

```
var container = cb.Build();

var button = container.Resolve<Button>();
var logging = container.Resolve<Logging>();

button.Fire(1); // Button clicked 1 times
button.Fire(2); // Button clicked 2 times
```

In a similar fashion, you could implement declarative subscriptions using attributes instead. And if you don't use Autofac, don't worry: most popular IoC containers are capable of implementing these sorts of declarative event wireups.

Summary

Generally speaking, we could have avoided discussing the Observer pattern in C# because the pattern itself is baked right into the language. That said, I have demonstrated some of the practical uses of the Observer (property change notifications) as well as some of the issues related to that (dependent properties). Furthermore, we looked at the way in which the Observer pattern is supported for reactive streams.

Thread safety is one concern when it comes to Observer, whether we are talking about individual events or entire collections. The reason why it shows up is because several observers on one component form a list (or similar structure) and then the question immediately arises as to whether that list is thread-safe and what exactly happens when it's being modified and iterated upon (for purposes of notification) at the same time.

State

I must confess: my behavior is governed by my state. If I didn't get enough sleep, I'm going to be a bit tired. If I had a drink, I wouldn't get behind the wheel. All of these are *states* and they govern my behavior: how I feel, what I can and cannot do.

I can, of course, transition from one state to another. I can go get a coffee, and this will take me from sleepy to alert (I hope!). So we can think of coffee as a *trigger* that causes a transition of yours truly from sleepy to alert. Here, let me clumsily illustrate it for you:[40]

```
       coffee
sleepy --------> alert
```

So, the State design pattern is a very simple idea: state controls behavior, state can be changed, the only thing which the jury is out on is *who* triggers the change from one state to another.

There are two ways in which we can model states:

- States are actual classes with behaviors, and these behaviors cause the transition from this state to another. In other words, a state's members are the options in terms of where we can go from that state.
- States and transitions are just enumerations. We have a special component called a *state machine* that performs the actual transitions.

Both of these approaches are viable, but it's really the second approach that is the most common. We'll take a look at both of them, but I must warn that I'll glance over the first one, since this isn't how people typically do things.

[40] I lied twice. First, I don't drive a car, I prefer electric bikes. Not that it affects drinking in any way – still not allowed. Second, I don't drink coffee.

State Driven State Transitions

We'll begin with the most trivial example out there: a light switch that can only be in the *on* and *off* states. The reason why we are choosing such a simple domain is that I want to highlight the madness (there's no other word for it) that a classic implementation of State brings, and this example is simple enough to do so without generating pages of code listings.

We are going to build a model where any state is capable of switching to some other state: this reflects the 'classic' implementation of the State design pattern (as per GoF book). First, let's model the light switch. All it has is a state and some means of switching from one state to another:

```
public class Switch
{
  public State State = new OffState();
}
```

This all looks perfectly reasonable, we have a switch that's in some state (either *on* or *off*). We can now define the `State` which, in this particular case, is going to be an actual class.

```
public abstract class State
{
  public virtual void On(Switch sw)
  {
    Console.WriteLine("Light is already on.");
  }

  public virtual void Off(Switch sw)
  {
    Console.WriteLine("Light is already off.");
  }
}
```

This implementation is far from intuitive, so much so that we need to discuss it slowly and carefully, because from the outset, nothing about the `State` class makes sense.

While the `State` is abstract (meaning you cannot instatiate it), it has non-abstract members that allow the switching from one state to another. This… to a reasonable person, it makes no sense. Imagine the light switch: it's the switch that changes states. The state itself isn't expected to change *itself*, and yet it appears this is exactly what it does.

Perhaps most bewildering, though, the default behavior of `State.On()`/`Off()` claims that we are *already* in this state! Note that these methods are virtual. This will come together, somewhat, as we implement the rest of the example.

We now implement the On and Off states:

```
public class OnState : State
{
  public OnState()
  {
    Console.WriteLine("Light turned on.");
  }
  public override void Off(Switch sw)
  {
    Console.WriteLine("Turning light off...");
    sw.State = new OffState();
  }
}
// similarly for OffState
```

The constructor of each state simply informs us that we have *completed* the transition. But the transition itself happens in `OnState.Off()` and `OffState.On()`. That is where the switching happens.

We can now complete the `Switch` class by giving it methods to actually switch the light on and off:

```
public class Switch
{
  public State State = new OffState();
  public void On()  { State.On(this);  }
  public void Off() { State.Off(this); }
}
```

So, putting it all together, we can run the following scenario:

```
LightSwitch ls = new LightSwitch(); // Light turned off
ls.On();  // Switching light on...
          // Light turned on
ls.Off(); // Switching light off...
          // Light turned off
ls.Off(); // Light is already off
```

Here is an illustrated transition from `OffState` to `OnState`:

```
          LightSwitch.On() -> OffState.On()
OffState ------------------------------------> OnState
```

On the other hand, the transition from `OnState` to `OnState` uses the base `State` class, the one that tells you that you are already in that state:

```
         LightSwitch.On() -> State.On()
OnState --------------------------------> OnState
```

Let me be the first to say that the implementation presented here is *terrible*. While being a nice demonstration of OOP equilibristics, it is an unreadable, unintuitive mess that goes against everything we learn about both OOP generally and Design Patterns in particular, specifically:

- A state typically does not switch itself.
- A list of possible transitions should not appear all over the place; it's best to keep it in one place (SRP).
- There is no need to have actual classes modeling states unless they have class-specific behaviors; this example could be reduced to something much simpler.

Maybe we should be enums to begin with?

Handmade State Machine

Let's try to define a state machine for a typical phone conversation.

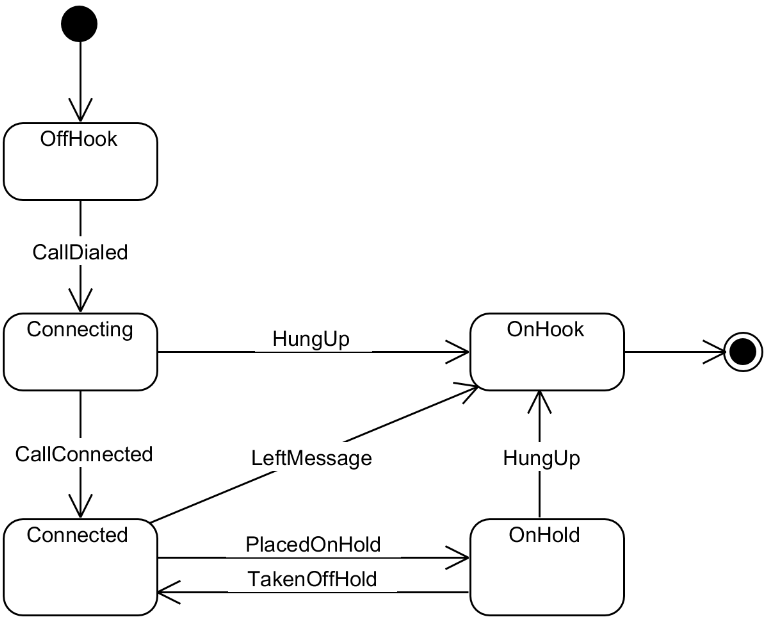
First of all, we'll describe the states of a phone:

```
public enum State
{
  OffHook,
  Connecting,
  Connected,
  OnHold
}
```

We can now also define transitions between states, also as an `enum`:

```
public enum Trigger
{
  CallDialed,
  HungUp,
  CallConnected,
  PlacedOnHold,
  TakenOffHold,
  LeftMessage
}
```

Now, the exact *rules* of this state machine, i.e., what transitions are possible, need to be stored somewhere. Here is a UML State Machine diagram showing what kind of transitions we want:



Let's use a dictionary of state-to-trigger/state pairs:

```
private static Dictionary<State, List<(Trigger, State)>> rules
  = new Dictionary<State, List<(Trigger, State)>>() { /* todo */ }
```

This is a little clumsy, but essentially the key of the dictionary is the `State` we're moving *from*, and the value is a list of `Trigger-State` pairs representing possible triggers while in this state and the state you move into when you use the trigger.

Let's initialize this data structure:

```
private static Dictionary<State, List<(Trigger, State)>> rules
  = new Dictionary<State, List<(Trigger, State)>>
  {
    [State.OffHook] = new List<(Trigger, State)>
    {
      (Trigger.CallDialed, State.Connecting)
    },
    [State.Connecting] = new List<(Trigger, State)>
    {
      (Trigger.HungUp, State.OffHook),
      (Trigger.CallConnected, State.Connected)
    },
    // more rules here
  };
```

We also need a starting (current) state, and we can also add an exit (terminal) state if we want the state machine to stop executing once that state is reached:

```
State state = State.OffHook, exitState = State.OnHook;
```

So in the above line, we start out with the `OffHook` state (when you're ready to make the call) and the exit state is when the phone is placed OnHook and the call is finished.

Having made this, we don't necessarily have to build a separate component for actually running (we use the term *orchestrating*) a state machine. For example, if we wanted to build an interactive model of the telephone, we could do it like this:

```
do
{
  Console.WriteLine($"The phone is currently {state}");
  Console.WriteLine("Select a trigger:");

  for (var i = 0; i < rules[state].Count; i++)
  {
    var (t, _) = rules[state][i];
    Console.WriteLine($"{i}. {t}");
  }

  int input = int.Parse(Console.ReadLine());

  var (_, s) = rules[state][input];
  state = s;
} while (state != exitState);
Console.WriteLine("We are done using the phone.");
```

The algorithm is fairly obvious: we let the user select one of the available triggers on the current state and, provided the trigger is valid, we transition to the right state by using that `rules` dictionary that we created earlier.

If the state we've reached is the exit state, we just jump out of the loop. Here's a sample interaction with the program:

```
The phone is currently OffHook
Select a trigger:
0. CallDialed
0
The phone is currently Connecting
Select a trigger:
0. HungUp
1. CallConnected
1
The phone is currently Connected
Select a trigger:
0. LeftMessage
1. HungUp
2. PlacedOnHold
2
The phone is currently OnHold
Select a trigger:
0. TakenOffHold
1. HungUp
1
We are done using the phone.
```

This hand-rolled state machine's main benefit is that it is very easy to understand: states and transitions are ordinary enumerations, the set of transitions is defined in a `Dictionary`, the start and end states are simple variables. I'm sure you'll agree this is much easier to understand than the example we started the chapter with.

Switch-Based State Machine

In our exploration of state machines, we have progressed from the needlessly-complicated Classic example where states are represented by classes, to a hand-crafted example where states are represented as enumerations, and now we shall experience one final step of degradation, as we stop using dedicated data types for states altogether.

But our simplifications won't end there: instead of jumping from one method call to another, we'll confine ourselves to an infinitely-recurring `switch` statement where state will be examined and transitions will happen by virtue of the state changing.

The scenario I want you to consider is a combination lock. The lock has a 4-digit code (e.g., `1234`) that you enter one digit at a time. As you enter the code, if you make a mistake, you get the `"FAILED"` output, but if you enter all digits correctly, you get `"UNLOCKED"` instead and you exit the state machine.

The entire scenario can fit into a single listing:

```
string code = "1234";
var state = State.Locked;
var entry = new StringBuilder();

while (true)
{
  switch (state)
  {
    case State.Locked:
      entry.Append(Console.ReadKey().KeyChar);

      if (entry.ToString() == code)
      {
        state = State.Unlocked;
        break;
      }

      if (!code.StartsWith(entry.ToString()))
      {
        // the code is blatantly wrong
        state = State.Failed;
      }
      break;
    case State.Failed:
      Console.CursorLeft = 0;
      Console.WriteLine("FAILED");
      entry.Clear();
      state = State.Locked;
      break;
    case State.Unlocked:
      Console.CursorLeft = 0;
      Console.WriteLine("UNLOCKED");
      return;
  }
}
```

As you can see, this is very much a state machine, albeit one that lacks any strucure. You couldn't examine it from the top level and be able to tell what all the possible states and transitions are. It is not clear, unless you really examine the code, how the transitions happen – and we're lucky there are no goto statements here to make jumps between the cases!

This Switch-Based State Machine approach is viable for scenarios with very small numbers of states and transitions. It loses out on structure, readability and maintainability, but can work as a quick patch if you do need a state machine quickly and are too lazy to make enum cases.

Overall, this approach does not scale and is difficult to manage, so I would not recommend it in production code. The only exception would be if such a machine was made using code generation on the basis of some external model.

Encoding Transitions with Switch Expressions

The switch-based state machine may be unwieldy, but that's partly due to the way information about states and transitioned is structured (because it's not). But there is a different kind of switch – a switch *statement* (as opposed to *expression*) that, thanks to pattern matching, allows us to neatly define state transitions.

Okay, time for a simple example. You're on a hunt for treasure and find a treasure chest that you can open or close… unless it's locked, in which case the situation is a bit more complicated (you need to have a key to lock or unlock a chest). Thus, we can encode the states and possible transitions as follows:

```
enum Chest
{
  Open, Closed, Locked
}

enum Action
{
  Open, Close
}
```

With this definition, we can write a method called `Manipulate` that takes us from one state to another. The general rules of chest operation are as follows:

- If the chest is locked, you can only open it if you have the key.
- If the chest is open and you close it while having the key, you lock it.
- If the chest is open and you don't have the key, you just close it.
- A closed (but not locked) chest can be opened whether you have the key or not.

The set of possible transitions can be encoded right in the structure of the pattern matching expression. Without further ado, here it is:

```
static Chest Manipulate(Chest chest,
  Action action, bool haveKey) =>
  (chest, action, haveKey) switch
  {
    (Chest.Closed, Action.Open, _) => Chest.Open,
    (Chest.Locked, Action.Open, true) => Chest.Open,
    (Chest.Open, Action.Close, true) => Chest.Locked,
    (Chest.Open, Action.Close, false) => Chest.Closed,

    _ => chest
  };
```

This approach has a number of advantages and disadvantages. The advantages are that:

- This state machine is easy to read.

- Guard conditions such as `haveKey` are easy to incorporate, and fit nice into pattern matching.

There are disadvantages, too:

- The formal set of rules for this state machine is defined in a way that cannot be extracted. There's no data store where the rules are kept, so you cannot generate a report or a diagram, or run any verification checks beyond those that are done by the compiler (it checks for exhaustiveness).
- If you need any behavior, such as state entry or exit behavior, this cannot be done easily in a switch expression â€" you would need to define a good old-fashioned method with a switch statement in it.

To sum up, this approach is great for simple state machines because it results in very readable code. But it's not exactly an 'enterprise' solution.

State Machines with Stateless

While hand-rolling a state machine works for the simplest of cases, you probably want to leverage an industrial-strength state machine framework. That way, you get a tested library with a lot more functionality. It's also fitting because we need to discuss additional state machine-related concepts, and implementing them by hand is rather tedious.

Before we move on to the concepts I want to discuss, let us first of all reconstruct our previous phone call example using Stateless[41]. Assuming the existence of the same enumerations `State` and `Trigger` as before, the definition of a state machine is very simple:

[41]Stateless can be found at https://github.com/dotnet-state-machine/stateless. It's worth noting that the phone call example actually comes from the authors of SimpleStateMachine, a project on which Stateless is based.

```
var call = new StateMachine<State, Trigger>(State.OffHook);

phoneCall.Configure(State.OffHook)
  .Permit(Trigger.CallDialed, State.CallConnected);

// and so on, then, to cause a transition, we do

call.Fire(Trigger.CallDialed); // call.State is now State.CallConnected
```

As you can see, Stateless' `StateMachine` class is a builder with a fluent interface. The motivation behind this API design will become apparent as we discuss the different intricacies of Stateless.

Types, Actions and Ignoring Transitions

Let's talk about the many features of Stateless and state machines generally.

First and foremost, Stateless supports states and triggers of *any* .NET type - it's not constrained to enums. You can use strings, numbers, anything you want. For example, a light switch could use a `bool` for states (`false` = off, `true` = on); we'll keep using enums for triggers. Here is how one would implement the `LightSwitch` example:

```
enum Trigger { On, Off }
var light = new StateMachine<bool, Trigger>(false);

light.Configure(false)           // if the light is off...
  .Permit(Trigger.On, true)      // we can turn it on
  .Ignore(Trigger.Off);          // but if it's already off we do nothing

// same for when the light is on
light.Configure(true)
  .Permit(Trigger.Off, false)
  .Ignore(Trigger.On)
  .OnEntry(() => timer.Start())
  .OnExit(() => timer.Stop());  // calculate time spent in this state

light.Fire(Trigger.On);  // Turning light on
light.Fire(Trigger.Off); // Turning light off
light.Fire(Trigger.Off); // Light is already off!
```

There are a few interesting things worth discussing here. First of all, this state machine has *actions* – things that happen as we enter a particular state. These are defined in `OnEntry()`, where you can provide a lambda that does something; similarly, you could invoke something at the moment the state is exited using `OnExit()`. One use of such transition actions would be to start a timer when entering a transition and stop it when exiting one, which could be used for tracking the amount of time spent in each state. For example, you might want to measure the time the light stays on for purposes of verifying electricity costs.

Another thing worth noting is the use of `Ignore()` builder methods. This basically tells the state machine to ignore the transition completely: if the light is already off, and we try to switch it off (as in the last line of the above listing), we instruct the state machine to simply ignore it, so no output is produced in that case.

Why is this important? Because, if you forget to `Ignore()` this transition or fail to specify it explicitly, Stateless will throw an `InvalidOperationException`:

> 'No valid leaving transitions are permitted from state 'False' for trigger 'False'. Consider ignoring the trigger.'

Reentrancy Again

Another alternative to the 'redundant switching' conundrum is Stateless' support for reentrant states. To replicate the example at the start of this chapter, we can configure the state machine so that, in the case of reentry into a state (meaning that we transition, say, from `false` to `false`), an action is invoked. Here is how one would configure it:

```
var light = new StateMachine<bool, Trigger>(false);

light.Configure(false) // if the light is off...
  .Permit(Trigger.On, true)  // we can turn it on
  .OnEntry(transition =>
  {
    if (transition.IsReentry)
      WriteLine("Light is already off!");
    else
      WriteLine("Turning light off");
  })
  .PermitReentry(Trigger.Off);

// same for when the light is on

light.Fire(Trigger.On);  // Turning light on
light.Fire(Trigger.Off); // Turning light off
light.Fire(Trigger.Off); // Light is already off!
```

In the above listing, `PermitReentry()` allows us to return to the `false` (off) state on a `Trigger.Off` trigger. Notice that, in order to output a corresponding message to the console, we use a different lambda: one that has a `Transition` parameter. The parameter has public members that describe the transition fully. This includes `Source` (the state we're transitioning from), `Destination` (the state we're going to), `Trigger` (what caused the transition) and `IsReentry`, a Boolean flag that we use to determine if this is a reentrant transition.

Hierarchical States

In the context of a phone call, it can be argued that the `OnHold` state is a substate of the `Connected` state, implying that when we're on hold, we're also connected. Stateless lets us configure the state machine like this:

```
phoneCall.Configure(State.OnHold)
    .SubstateOf(State.Connected)
    // etc.
```

Now, if we are in the `OnHold` state; `phoneCall.State` will give us `OnHold`, but there's also a `phoneCall.IsInState(State)` method that will return `true` when called with either `State.Connected` or `State.OnHold`.

More Features

Let's talk about a few more features related to state machines that are implemented in Stateless.

- **Guard clauses** allow you to enable and disable transitions at will by calling `PermitIf()` and providing `bool`-returning lambda functions, e.g.:

```
phoneCall.Configure(State.OffHook)
  .PermitIf(Trigger.CallDialled, State.Connecting, () => IsValidNumber)
  .PermitIf(Trigger.CallDialled, State.Beeping, () => !IsValidNumber);
```

- **Parameterized triggers** are an interesting concept. Essentially, you can attach parameters to triggers such that, in addition to the trigger itself, there's also additional information that can be passed along. For example, if a state machine needs to notify a particular employee, you can specify an email to be used for notification:

```
var notifyTrigger = workflow.SetTriggerParameters<string>(Trigger.Notify);
workflow.Configure(State.Notified)
  .OnEntryFrom(assignTrigger, email => SendEmail(email));
workflow.Fire(notifyTrigger, "foo@bar.com");
```

- **External storage** is a feature of Stateless that lets you store the internal state of a state machine externally (e.g., in a database) instead of using the `StateMachine` class itself. To use it, you simply define the getter and setter methods in the `StateMachine` constructor:

```
var stateMachine = new StateMachine<State, Trigger>(
  () => database.ReadState(),
  s => database.WriteState(s));
```

- **Introspection** allows us to actually look at the table of triggers that can be fired from the current state through `PermittedTriggers` property.

This is far from an exhaustive list of features that Stateless offers, but it covers all the important parts.

Summary

As you can see, the whole business of state machines extends way beyond simple transitions: it allows a lot of complexity to handle the most demanding business cases. Let us recap some of the state machine features we've discussed:

- State machines involve two collections: states and triggers. States model the possible states of the system and triggers transition us from one state to another. You are not limited to enumerations: you can use ordinary data types.
- Attempting a transition that's not configured will result in an exception.
- It is possible to explicitly configure entry and exit actions for each state.
- Reentrancy can be explicitly allowed in the API and, furthermore, you can determine whether or not a reentry is occuring in the entry/exit action.
- Transitions can be turned on an off through guard conditions. They can also be parameterized.
- States can be hierarchical, i.e, they can be substates of other states. An additional method is then required to determine whether you're in a particular (parent) state.

While most of the above might seem like overengineering, these features provide great flexibility in defining real-world state machines.

Strategy

Suppose you decide to take an array or vector of several strings and output them as a list,

- just
- like
- this

If you think about the different output formats, you probably know that you need to take each element and output it with some additional markup. But in the case of languages such as HTML or LaTeX, the list will also need the start and end tags or markers.

We can formulate a strategy for rendering a list:

- Render the opening tag/element
- For each of the list items, render that item
- Render the closing tag/element

Different strategies can be formulated for different output formats, and these strategies can be then fed into a general, non-changing algorithm to generate the text.

This is yet another pattern that exists in dynamic (runtime-replaceable) and static (generics-based, fixed) incarnations. Let's take a look at both of them.

Dynamic Strategy

So our goal is to print a simple list of text items in the following formats:

```
public enum OutputFormat
{
  Markdown,
  Html
}
```

The skeleton of our strategy will be defined in the following base class:

```
public interface IListStrategy
{
  void Start(StringBuilder sb);
  void AddListItem(StringBuilder sb, string item);
  void End(StringBuilder sb);
}
```

Now let us jump to our text processing component. This component would have a list-specific method called, say, `AppendList()`.

```
public class TextProcessor
{
  private StringBuilder sb = new StringBuilder();
  private IListStrategy listStrategy;

  public void AppendList(IEnumerable<string> items)
  {
    listStrategy.Start(sb);
    foreach (var item in items)
      listStrategy.AddListItem(sb, item);
    listStrategy.End(sb);
  }

  public override string ToString() => sb.ToString();
}
```

So we've got a buffer called `sb` where all the output goes, the `listStrategy` that we're using for rendering lists, and of course `AppendList()` which specifies the set of steps that need to be taken to actually render a list with a given strategy.

Now, pay attention here. Composition, as used above, is one of two possible options that can be taken to allow concrete implementations of a skeleton algorithm. Instead, we could add functions such as AddListItem() as abstract or virtual members to be overridden by derived classes: that's what the Template Method pattern does.

Anyways, back to our discussion. We can now go ahead and implement different strategies for lists, such as a HtmlListStrategy:

```
public class HtmlListStrategy : IListStrategy
{
  public void Start(StringBuilder sb) => sb.AppendLine("<ul>");
  public void End(StringBuilder sb) => sb.AppendLine("</ul>");

  public void AddListItem(StringBuilder sb, string item)
  {
    sb.AppendLine($"  <li>{item}</li>");
  }
}
```

By implementing the overrides, we fill in the gaps that specify how to process lists. We implement a MarkdownListStrategy in a similar fashion, but because Markdown does not need opening/closing tags, we only do work in the AddListItem() method:

```
public class MarkdownListStrategy : IListStrategy
{
  // markdown doesn't require list start/end tags
  public void Start(StringBuilder sb) {}
  public void End(StringBuilder sb) {}

  public void AddListItem(StringBuilder sb, string item)
  {
    sb.AppendLine($" * {item}");
  }
}
```

We can now start using the TextProcessor, feeding it different strategies and getting different results. For example:

```
var tp = new TextProcessor();
tp.SetOutputFormat(OutputFormat.Markdown);
tp.AppendList(new []{"foo", "bar", "baz"});
WriteLine(tp);

// Output:
// * foo
// * bar
// * baz
```

We can make provisions for strategies to be switchable at runtime – this is precisely why we call this implementation a *dynamic* strategy. This is done in the `SetOutputFormat()` method, whose implementation is trivial:

```
public void SetOutputFormat(OutputFormat format)
{
  switch (format) {
    case OutputFormat.Markdown:
      listStrategy = new MarkdownListStrategy();
      break;
    case OutputFormat.Html:
      listStrategy = new HtmlListStrategy();
      break;
    default:
      throw new ArgumentOutOfRangeException(nameof(format), format, null);
  }
}
```

Now, switching from one strategy to another is trivial, and you get to see the results straight away:

```
tp.Clear(); // erases underlying buffer
tp.SetOutputFormat(OutputFormat.Html);
tp.AppendList(new[] { "foo", "bar", "baz" });
WriteLine(tp);

// Output:
// <ul>
//   <li>foo</li>
//   <li>bar</li>
//   <li>baz</li>
// </ul>
```

Static Strategy

Thanks to the magic of generics, you can bake any strategy right into the type. Only minimal changes are necessary to the `TextStrategy` class:

```
public class TextProcessor<LS>
  where LS : IListStrategy, new()
{
  private StringBuilder sb = new StringBuilder();
  private IListStrategy listStrategy = new LS();

  public void AppendList(IEnumerable<string> items)
  {
    listStrategy.Start(sb);
    foreach (var item in items)
      listStrategy.AddListItem(sb, item);
    listStrategy.End(sb);
  }

  public override string ToString() => return sb.ToString();
}
```

What changed in the dynamic implementation is as follows: we added the `LS` generic argument, made a `listStrategy` member with this type, and started using it instead of the reference we had previously. The results of calling the adjusted AppendList() are identical to what we had before.

```
var tp = new TextProcessor<MarkdownListStrategy>();
tp.AppendList(new []{"foo", "bar", "baz"});
WriteLine(tp);

var tp2 = new TextProcessor<HtmlListStrategy>();
tp2.AppendList(new[] { "foo", "bar", "baz" });
WriteLine(tp2);
```

The output from the above example is the same as for the dynamic strategy. Note that we've had to make two instances of TextProcessor, each with a distinct list-handling strategy, since it is impossible to switch a type's strategy mid-stream: it is baked right into the type.

Equality and Comparison Strategies

The most well-known use of the Strategy pattern inside .NET is, of course, the use of equality and comparison strategies.

Consider a simple class such as the following:

```
class Person
{
  public int Id;
  public string Name;
  public int Age;
}
```

As it stands, you can put several Person instances inside a List, but calling Sort() on such a list would be meaningless.

```
var people = new List<Person>();
people.Sort(); // does not do what you want
```

The same goes for comparisons using the == and != operators: at the moment, all those comparisons would do is a reference-based comparison.

We need to clearly distinguish two types of operations:

- Equality checks whether or not two instances of an object are equal, according to the rules you define. This is converned by the `IEquatable<T>` interface (the `Equals()` method) as well as operators == and != which typically use the `Equals()` method internally.
- Comparison allows you to compare two objects and find which one is less, equal, or greater than another. This is covered by the `IComparable<T>` interface and is required for things like sorting.

By implementing `IEquatable<T>` and `IComparable<T>`, every object can expose its own comparison and equality strategies. For example, if we assume that people have unique `Id`s, we can use that id value for comparison:

```
public int CompareTo(Person other)
{
  if (ReferenceEquals(this, other)) return 0;
  if (ReferenceEquals(null, other)) return 1;
  return Id.CompareTo(other.Id);
}
```

So now, calling `people.Sort()` makes a kind of sense – it will use the built-in `CompareTo()` method that we've written. But there is a problem: a typical class can only have one default `CompareTo()` implementation for comparing the class with itself. The same goes for equality. So what if your comparison strategy changes at runtime?

Luckily, BCL designers have thought of that, too. We can specify the comparison strategy right at the call site, simply by passing in a lambda:

```
people.Sort((x, y) => x.Name.CompareTo(y.Name));
```

This way, even though `Person`'s default comparison behavior is to compare by id, we can compare by name if we need to.

But that's not all! There is a third way in which a comparison strategy can be defined. This way is useful if some strategies are common and you want to preserve them inside the class itself.

The idea is this: you define a nested class that implements the `IComparer<T>` interface. You then expose this class as a static variable:

```
public class Person
{
  // ... other members here
  private sealed class NameRelationalComparer : IComparer<Person>
  {
    public int Compare(Person x, Person y)
    {
      if (ReferenceEquals(x, y)) return 0;
      if (ReferenceEquals(null, y)) return 1;
      if (ReferenceEquals(null, x)) return -1;
      return string.Compare(x.Name, y.Name,
        StringComparison.Ordinal);
    }
  }

  public static IComparer<Person> NameComparer { get; }
    = new NameRelationalComparer();
}
```

As you can see, the above class defines a standalone strategy for comparing two `Person` instances using their names. We can now simply take a static instance of this class and feed it into the `Sort()` method:

```
people.Sort(Person.NameComparer);
```

As you may have guessed, the situation with equality comparison is fairly similar: you can use an `IEquatable<T>`, pass in a lambda or generate a class that implements an `IEqualityComparer<T>`. Your choice!

Functional Strategy

The functional variation of the Strategy pattern is simple: all OOP constructs are simply replaced by functions. First of all, `TextProcessor` devolves from being a class to being a function. This is actually idiomatic (i.e., the right thing to do) because `TextProcessor` has a single operation.

```
let processList items startToken itemAction endToken =
  let mid = items |> (Seq.map itemAction) |> (String.concat "\n")
  [startToken; mid; endToken] |> String.concat "\n"
```

The above function takes 4 arguments: a sequence of items, the starting token (note: it's a token, not a function), a function for processing each element, and the ending token. Since this is a function, this approach assumes that `processList` is stateless, i.e., it does not keep any state internally.

As you can see from the above, our strategy is not just a single, neatly self-contained element, but rather a combination of three different items: the start and end tokens as well as a function that operates upon each of the elements in the sequence. We can now specialize `processList` in order to implement HTML and Markdown processing as before:

```
let processListHtml items =
  processList items "<ul>" (fun i -> "  <li>" + i + "</li>") "</ul>"

let processListMarkdown items =
  processList items "" (fun i -> " * " + i) ""
```

This is how you would use these specializations, with predictable results:

```
let items = ["hello"; "world"]
printfn "%s" (processListHtml items)
printfn "%s" (processListMarkdown items)
```

The interesting thing to note about this example is that the interface of `processList` gives absolutely no hints whatsoever as to what the client is supposed to provide as the `itemAction`. All they know it's an `'a -> string`, so we rely on them to guess correctly what it's actually for.

Summary

The Strategy design pattern allows you to define a skeleton of an algorithm and then use composition to supply the missing implementation details related to a particular strategy. This approach exists in two incarnations:

- *Dynamic strategy* simply keeps a reference to the strategy currently being used. Want to change to a different strategy? Just change the reference. Easy!
- *Static strategy* requires that you choose the strategy at compile time and stick with it – there is no scope for changing your mind later on.

Should one use dynamic or static strategies? Well, dynamic ones allow you reconfiguration of the objects after they have been constructed. Imagine a UI setting which controls the form of the textual output: what would you rather have, a switchable `TextProcessor` or two variables of type `TextProcessor<MarkdownStrategy` and `TextProcessor<HtmlStrategy>`? It’s really up to you.

Template Method

The Strategy and Template Method design patterns are very similar, so much so that, just like with Factories, I would be very tempted to merge those patterns into a single Skeleton Method design pattern. I will resist the urge.

The difference between Strategy and Template Method is that Strategy uses composition (whether static or dynamic) whereas Template Method uses inheritance. But the core principle of defining the skeleton of an algorithm in one place and its implementation details in other places remain, once again observing OCP (we simply *extend* systems).

Game Simulation

Most board games are very similar: the game starts (some sort of set-up takes place), players take turns until a winner is decided, and then the winner can be announced. It doesn't matter what the game is – chess, checkers, something else, we can define the algorithm as follows:

```
public abstract class Game
{
  public void Run()
  {
    Start();
    while (!HaveWinner)
      TakeTurn();
    WriteLine($"Player {WinningPlayer} wins.");
  }
}
```

As you can see, the `run()` method, which runs the game, simply uses a set of other methods and properties. Those methods are abstract, and also have `protected` visibility so they don't get called from the outside:

```
protected abstract void Start();
protected abstract bool HaveWinner { get; }
protected abstract void TakeTurn();
protected abstract int WinningPlayer { get; }
```

To be fair, some of the above members, especially `void`-returning ones, do not necessarily have to be abstract. For example, if some games have no explicit `start()` procedure, having `start()` as abstract violates ISP since members which do not need it would still have to implement it. In the Strategy chapter we deliberately made an interface, but with Template Method, the case is not so clear-cut.

Now, in addition to the above, we can have certain protected fields that are relevant to all games: the number of players and the index of the current player:

```
public abstract class Game
{
  public Game(int numberOfPlayers)
  {
    this.numberOfPlayers = numberOfPlayers;
  }

  protected int currentPlayer;
  protected readonly int numberOfPlayers;
  // other members omitted
}
```

From here on out, the Game class can be extended to implement a game of chess:

```
public class Chess : Game
{
  public Chess() : base(2) { /* 2 players */ }

  protected override void Start()
  {
    WriteLine($"Starting a game of chess with {numberOfPlayers} players.");
  }
  protected override bool HaveWinner => turn == maxTurns;

  protected override void TakeTurn()
  {
    WriteLine($"Turn {turn++} taken by player {currentPlayer}.");
    currentPlayer = (currentPlayer + 1) % numberOfPlayers;
  }

  protected override int WinningPlayer => currentPlayer;

  private int maxTurns = 10;
  private int turn = 1;
}
```

A game of chess involves 2 players, so that's the value fed into the base class' constructor. We then proceed to override all the necessary methods, implementing some very simple simulation logic for ending the game after 10 turns. We can now use the class with `new Chess().Run()` – here is the output:

```
Starting a game of chess with 2 players
Turn 0 taken by player 0
Turn 1 taken by player 1
...
Turn 8 taken by player 0
Turn 9 taken by player 1
Player 0 wins.
```

And that's pretty much all there is to it!

Functional Template Method

As you may have guessed, the functional approach to Template Method is to simply define a standalone function `runGame()` that takes the templated parts as parameters. The only problem is that a game is an *inherently mutable* scenario, which means we have to have some sort of container representing the state of the game. We can try using a record type:

```
type GameState = {
  CurrentPlayer: int;
  NumberOfPlayers: int;
  HaveWinner: bool;
  WinningPlayer: int;
}
```

With this set-up, we end up having to pass an instance of `GameState` into every function that is part of the template method. The method itself, mind you, is rather simple:

```
let runGame initialState startAction takeTurnAction haveWinnerAction =
  let state = initialState
  startAction state
  while not (haveWinnerAction state) do
    takeTurnAction state
  printfn "Player %i wins." state.WinningPlayer
```

The implementation of a chess game isn't a particularly difficult affair either, the only real problem being the initialization and modification of internal state:

```
let chess() =
  let mutable turn = 0
  let mutable maxTurns = 10
  let state = {
    NumberOfPlayers = 2;
    CurrentPlayer = 0;
    WinningPlayer = -1;
  }
  let start state =
    printfn "Starting a game of chess with %i players" state.NumberOfPlayers

  let takeTurn state =
    printfn "Turn %i taken by player %i." turn state.CurrentPlayer
    state.CurrentPlayer <- (state.CurrentPlayer+1) % state.NumberOfPlayers
    turn <- turn + 1
    state.WinningPlayer <- state.CurrentPlayer
  let haveWinner state =
    turn = maxTurns

  runGame state start takeTurn haveWinner
```

So, just to recap, what we’re doing here is initializing all the functions required by the method/function right inside the outer function (this is completely legitimate in both C# and F#) and then pass each of those functions into `runGame`. Notice also that we have some mutable state that is used throughout the sub-function calls.

Overall, implementation of Template Method using functions instead of objects is quite possible if you’re prepared to introduce record types and mutability into your code. And sure, theoretically, you could rewrite this example and get rid of mutable state by essentially storing a snapshot of each game state and passing that in a recursive setting â€” this would effectively turn a Template Method into a kind of templated State pattern. Try it!

Summary

Unlike the Strategy, which uses composition and thus branches into static and dynamic variations, Template Method uses inheritance and, as a consequence, it can only be static, since there is no way to manipulate the inheritance characteristics of an object once it’s been constructed.

The only design decision in a Template Method is whether you want the methods used by the Template Method to be abstract or actually have a body, even if that body is empty. If you foresee some methods unnecessary for *all* inheritors, go ahead and make them empty/non-abstract so as to comply with ISP.

Visitor

To explain this pattern I'm going to jump into an example first and then discuss the pattern itself. Hope you don't mind!

Suppose you have parsed a mathematical expression (with the use of the Interpreter pattern, of course!) composed of `double` values and addition operators, e.g.,

```
(1.0 + (2.0 + 3.0))
```

This expression can expressed using an object hierarchy similar to the following:

```
public abstract class Expression { /* nothing here (yet) */ }

public class DoubleExpression : Expression
{
  private double value;

  public DoubleExpression(double value) { this.value = value; }
}

public class AdditionExpression : Expression
{
  private Expression left, right;

  public AdditionExpression(Expression left, Expression right)
  {
    this.left = left;
    this.right = right;
  }
}
```

Given this set-up, you are interested in two things:

- Printing the OOP expression as text; and
- Evaluating the expression's value

You also want to do those two things (and many other possible operations on these trees) as uniformly and succintly as possible. How would you do it? Well, there are many ways and we'll take a look at them all, starting with the implementation of the printing operation.

Intrusive Visitor

The simplest solution is to take the base `Expression` class and add an abstract member to it.

```
public abstract class Expression
{
  // adding a new operation
  public abstract void Print(StringBuilder sb);
}
```

In addition to breaking OCP, this modification hinges on the assumption that you actually have access to the hierarchy's source code – something that's not always guaranteed. But we've got to start somewhere, right? So with this change, we need to implement `Print()` in `DoubleExpression` (that's easy, so I'll omit it here) as well as in `AdditionExpression`:

```
public class AdditionExpression : Expression
{
  ...
  public override void Print(StringBuilder sb)
  {
    sb.Append(value: "(");
    left.Print(sb);
    sb.Append(value: "+");
    right.Print(sb);
    sb.Append(value: ")");
  }
}
```

Ooh, this is fun! We are polymorphically and recursively calling `Print()` on subexpressions. Wonderful, let's test this out:

```
var e = new AdditionExpression(
  left: new DoubleExpression(1),
  right: new AdditionExpression(
    left: new DoubleExpression(2),
    right: new DoubleExpression(3)));
var sb = new StringBuilder();
e.Print(sb);
WriteLine(sb); // (1.0 + (2.0 + 3.0))
```

Well, this was easy. But now imagine you've got 10 inheritors in the hierarchy (not uncommon, by the way, in real-world scenarios) and you need to add some new `Eval()` operation. That's 10 modifications that need to be done in 10 different classes. But OCP isn't the real problem.

The real problem is SRP. You see, a problem such as printing is a special concern. Rather than stating that every expression should print itself, why not introduce an `ExpressionPrinter` that knows how to print expressions? And, later on, you can introduce an `ExpressionEvaluator` that knows how to perform the actual calculations. All without affecting the `Expression` hierarchy in any way.

Reflective Printer

Now that we've decided to make a *separate* printer component, let's get rid of `Print()` member functions (but keep the base class, of course).

```
abstract class Expression
{
  // nothing here!
};
```

Now let's try to implement an `ExpressionPrinter`. My first instinct would be to write something like this:

```
public static class ExpressionPrinter
{
  public static void Print(DoubleExpression e, StringBuilder sb)
  {
    sb.Append(de.Value);
  }

  public static void Print(AdditionExpression ae, StringBuilder sb)
  {
    sb.Append("(");
    Print(ae.Left, sb);  // will not compile!!!
    sb.Append("+");
    Print(ae.Right, sb); // will not compile!!!
    sb.Append(")");
  }
}
```

Odds of the above compiling: ZERO. C# knows that, say, `ae.Left` is an `Expression`, but since it doesn't check the type at runtime (unlike various dynamically typed languages), it doesn't know which overload to call. Too bad!

What can be done here? Well, only one thing – remove the overloads and check the type at runtime:

```
public static class ExpressionPrinter
{
  public static void Print(Expression e, StringBuilder sb)
  {
    if (e is DoubleExpression de)
    {
      sb.Append(de.Value);
    }
    else if (e is AdditionExpression ae)
    {
      sb.Append("(");
      Print(ae.Left, sb);
      sb.Append("+");
      Print(ae.Right, sb);
      sb.Append(")");
    }
  }
}
```

The above is actually a usable solution:

```
var e = new AdditionExpression(
  left: new DoubleExpression(1),
  right: new AdditionExpression(
    left: new DoubleExpression(2),
    right: new DoubleExpression(3)));
var sb = new StringBuilder();
ExpressionPrinter.Print(e, sb);
WriteLine(sb);
```

This approach has a fairly significant downside: there are no compiler checks that you *have*, in fact, implemented printing for every single element in the hierarchy. When a new element gets added, you can keep using `ExpressionPrinter` without modification, and it will just skip over any element of the new type.

But this is a viable solution. Seriously, it's quite possible to stop here and never go any further in the Visitor pattern: the `is` operator isn't *that* expensive and I think many developers will remember to cover every single type of object in that `if` statement.

Extension Methods?

You could be forgiven for thinking that the problem of separating out an `ExpressionPrinter` can be somehow solved without the use of type checks. Sadly, this set-up also devolves to the use of the Reflective Visitor.

Sure, you can take both `DoubleExpression` and `AdditionExpression` and give them `Print()` extension methods that would be callable directly on the object, while residing elsewhere. However, your implementation of `AdditionExpression.Print()` will still have several problems:

```
public static void Print(this AdditionExpression ae, StringBuilder sb)
{
  sb.Append("(");
  ae.Left.Print(sb); // oops
  sb.Append("+");
  ae.Right.Print(sb);
  sb.Append(")");
}
```

The first problem is that, since this is an extension method, we need to make the `Left` and `Right` members public so that the extension method can access them.

But that's not the real problem. The main issue here is that `ae.Left.Print()` cannot be called because `ae.Left` is a general `Expression`. How would you support it? Well, this is where you would devolve to a Reflective Printer by implementing an extension method on the root element of the hierarchy and perform type checks:

```
public static void Print(this Expression e, StringBuilder sb)
{
  switch (e)
  {
    case DoubleExpression de:
      de.Print(sb);
      break;
    case AdditionExpression ae:
      ae.Print(sb);
      break;
    // and so on
  }
}
```

This solution runs into the same problem as the original, namely the issue that there's no verification to ensure that *every* inheritor of `Expression` is covered by the `switch` statement. Now, admittedly, this is something that we can actually force, thus lending the Reflective Printer its true name by using… reflection!

Extension method classes are static, and can have both static fields and constructors, so we can map out all the inheritors and attempt to find the methods that handle them:

```
public static class ExpressionPrinter
{
  private static Dictionary<Type, MethodInfo> methods
    = new Dictionary<Type, MethodInfo>();

  static ExpressionPrinter()
  {
    var a = typeof(Expression).Assembly;
    var classes = a.GetTypes()
      .Where(t => t.IsSubclassOf(typeof(Expression)));
    var printMethods = typeof(ExpressionPrinter).GetMethods();
    foreach (var c in classes)
    {
      // find extension method that takes this class
      var pm = printMethods.FirstOrDefault(m =>
        m.Name.Equals(nameof(Print)) &&
        m.GetParameters()?[0]?.ParameterType == c);

      methods.Add(c, pm);
    }
  }
}
```

With this set-up, the extension method `Print()` implemented for the base type `Expression` now devolves to:

```
public static void Print(this Expression e, StringBuilder sb)
{
  methods[e.GetType()].Invoke(null, new object[] {e, sb});
}
```

Naturally, this approach has significant performance costs. There are ways to offset these costs, such as using `Delegate.CreateDelegate()` to avoid storing those `MethodInfo` objects and instead having ready-to-call delegates when the need arises.

Finally, there's always the 'nuclear option': generating code that creates those calls at runtime. Of course, this comes with its own set of problems: you'll be generating code either on the basis of reflection (which means that you're almost always one step behind, because you need a binary in order to extract type information), or, alternatively, you'll be inspecting actual written code using a parser framework provided by Roslyn, ReSharper, Rider or some similar mechanism.

Functional Reflective Visitor

It's worth noting that the approach adopted above is *precisely* the approach you would adopt in a language such as F#, the only difference being that, of course, instead of inheritance hierarchies you would mainly be dealing with functions.

If, instead of a hierarchy, you defined your expression types in a discriminated union such as

```
type Expression =
  | Add of Expression * Expression
  | Mul of Expression * Expression
  ...
```

then any visitor you would implement would, most likely, have a structure similar to the following:

```
let rec process expr =
  match expr with
  | And(lhs,rhs) -> ...
  | Mul(lhs,rhs) -> ...
  ...
```

This approach is precisely equivalent to the approach taken in our C# implementation. Each of the cases in a `match` expression would effectively be turned into an `is` check. There are major differences, however. First of all, concrete case filters and guard conditions in F# are easier to read than nested `if` statements in C#. The possible recursiveness of the entire process is a lot more expressive.

Improvements

While it's not possible to statically enforce the presence of every single necessary type check in the above example, it *is* possible to generate exceptions if the appropriate implementation is missing. To do this, simply make a dictionary that maps the supported types to lambda functions that process those types, i.e.:

```
private static DictType actions = new DictType
{
  [typeof(DoubleExpression)] = (e, sb) =>
  {
    var de = (DoubleExpression) e;
    sb.Append(de.Value);
  },
  [typeof(AdditionExpression)] = (e, sb) =>
  {
    var ae = (AdditionExpression) e;
    sb.Append("(");
    Print(ae.Left, sb);
    sb.Append("+");
    Print(ae.Right, sb);
    sb.Append(")");
  }
};
```

Now you can implement a top-level `Print()` method in a much simpler fashion. In fact, for bonus points, you can use the C# extension methods mechanic to add `Print()` as a method of any `Expression`:

```
public static void Print(this Expression e, StringBuilder sb)
{
  actions[e.GetType()](e, sb);
}
// sample use:
myExpression.Print(sb)
```

Whether or not you use extension methods or just ordinary static or instance methods on a `Printer` is intirely irrelevant for the purposes of SRP. Both an ordinary class and an extension-method class serve to isolate printing functionality from the data structures themselves, the only difference being whether or not you consider printing part of `Expression`'s API. Which I personally think is reasonable: I like the idea of `expression.Print()`, `expression.Eval()` and so on. Though if you are an OOP purist, you might hate this approach.

What is Dispatch?

Whenever people speak of visitors, the word *dispatch* is brought up. What it it? Well, put simply, ‘dispatch’ is a problem of figuring out which methods to call – specifically, how many pieces of information are required in order to make the call.

Here’s a simple example:

```
interface IStuff { }
class Foo : IStuff { }
class Bar : IStuff { }

public class Something
{
  static void func(Foo foo) { }
  static void unc(Bar bar) { }
}
```

Now, if I make an ordinary Foo object, I’ll have no problem calling `func()` with it:

```
Foo foo = new Foo();
func(foo); // this is fine
```

But if I decide to cast it to a base type (interface or class), the compiler will not know which overload to call:

```
Stuff stuff = new Foo;
func(stuff); // oops!
```

Now, let’s think about this polymorphically: is there *any* way we can coerce the system to invoke the correct overload without any runtime (`is`, `as` and similar) checks? Turns out there is.

See, when you call something on a `Stuff`, that call *can* be polymorphic and it can be dispatched right to the necessary component. Which in turn can call the necessary overload. This is called *double dispatch* because:

1. First you do a polymorphic call on the actual object.
2. Inside the polymorphic call, you call the overload. Since, inside the object, `this` has a precise type (e.g., a `Foo` or `Bar`), the right overload is triggered.

Here's what I mean:

```
interface Stuff {
  void call();
}
class Foo : Stuff {
  void call() { func(this); }
}
class Bar : Stuff {
  void call() { func(this); }
}

void func(Foo foo) {}
void func(Bar bar) {}
```

Can you see what's happening here? We cannot just stick one generic `call()` implemenetation into `Stuff`: the distinct implementations *must* be in their respective classes so that the `this` pointer is suitably typed.

This implementation lets you write the following:

```
Stuff foo = new Foo;
foo.call();
```

And here is a schematic showing what's going on:

```
            this = Foo
foo.call() ------------> func(foo)
```

Dynamic Visitor

Let's come back to the `ExpressionPrinter` example that I claimed has no zero chance of working:

```
public class ExpressionPrinter
{
  public void Print(AdditionExpression ae, StringBuilder sb)
  {
    sb.Append("(");
    Print(ae.Left, sb);
    sb.Append("+");
    Print(ae.Right, sb);
    sb.Append(")");
  }

  public void Print(DoubleExpression de, StringBuilder sb)
  {
    sb.Append(de.Value);
  }
}
```

What if I told you I could make it work as-is just by adding two keywords and raising the computational cost of the `Print(ae,sb)` method? I'm sure you can guess what I'm talking about already. Yeah, I'm talking about dynamic dispatch:

```
public void Print(AdditionExpression ae, StringBuilder sb)
{
  sb.Append("(");
  Print((dynamic)ae.Left, sb);  // <-- look closely here
  sb.Append("+");
  Print((dynamic)ae.Right, sb); // <-- and here
  sb.Append(")");
}
```

The whole business of `dynamic` was added to C# in order to support dynamically typed languages. One aspect of some of those languages is the ability to dynamically dispatch, i.e., to make call decisions at runtime as opposed to compile-time. And that's exactly what we're doing here!

Here's how you would call it:

```
var e = ...; // as before
var ep = new ExpressionPrinter();
var sb = new StringBuilder();
ep.Print((dynamic)e, sb); // <-- note the cast here
WriteLine(sb);
```

By casting a variable to `dynamic`, we defer dispatch decisions until runtime. Thus, we get the correct calls happening; there are only a few problems, namely:

- There is a fairly significant performance penalty related to this type of dispatching.
- If a needed method is missing, you will get a runtime error.
- You can run into serious problems with inheritance.

A dynamic visitor is a good solution if you expect the object graph you visit to be small, and the calls to be infrequent. Otherwise, the performance penalty might make the entire endeavor untenable.

Classic Visitor

The 'classic' implementation of the Visitor design pattern uses *double dispatch*. There are conventions as to what the visitor member functions are called:

- Methods of the visitor are typically called `Visit()`.
- Methods implemented throughout the hierarchy are typically called `Accept()`.

So now, once again, we have something to put into the base `Expression` class: the `Accept()` function.

```
public abstract class Expression
{
  public abstract void Accept(IExpressionVisitor visitor);
}
```

As you can see, the above refers to an interface type named `IExpressionVisitor` that can serve as a base type for various visitors such as `ExpressionPrinter`, `ExpressionEvaluator` and similar. Now, every single implementor of `Expression` is now *required* to implement `Accept()` in an identical way, specifically:

```
public override void Accept(IExpressionVisitor visitor)
{
  visitor.Visit(this);
}
```

On the surface of it, this looks like a violation of DRY (Don't Repeat Yourself), another self-descriptive principle. However, if you think about it, every implementor will have a differently typed `this` reference, so this is not another case of cut-and-paste programming that static analysis tools like to complain about so much.

Now, on the other side, we can define the `IExpressionVisitor` interface as follows:

```
public interface IExpressionVisitor
{
  void Visit(DoubleExpression de);
  void Visit(AdditionExpression ae);
}
```

Notice that we *absolutely must* define overloads for all expression objects; otherwise, we would get a compilation error when implementing the corresponding `Accept()`. We can now implement this interface to define our `ExpressionPrinter`:

```
public class ExpressionPrinter : IExpressionVisitor
{
  StringBuilder sb = new StringBuilder();

  public void Visit(DoubleExpression de)
  {
    sb.Append(de.Value);
  }

  public void Visit(AdditionExpression ae)
  {
    // wait for it!
  }

  public override string ToString() => sb.ToString();
}
```

The implementation for a `DoubleExpression` fairly obvious, but here's the implementation for an `AdditionExpression`:

```
public void Visit(AdditionExpression ae)
{
  sb.Append("(");
  ae.Left.Accept(this);
  sb.Append("+");
  ae.Right.Accept(this);
  sb.Append(")");
}
```

Notice how the calls now happen *on* the subexpressions themselves, leveraging double dispatch once again. As for the usage of the new double dispatch Visitor, here it is:

```
var e = new AdditionExpression(
  new DoubleExpression(1),
  new AdditionExpression(
    new DoubleExpression(2),
    new DoubleExpression(3)));
var ep = new ExpressionPrinter();
ep.Visit(e);
WriteLine(ep.ToString()); // (1 + (2 + 3))
```

Sadly, it's impossible to construct an extension-method analogue to the above implementation because, guess what, static classes cannot implement interfaces. If you want to hide the expression printer behind a slightly nicer API, you can go with the following:

```
public static class ExtensionMethods
{
  public void Print(this DoubleExpression e, StringBuilder sb)
  {
    var ep = new ExpressionPrinter();
    ep.Print(e, sb);
  }
  // other overloads here
}
```

Of course, it's up to you to implement all the correct overloads, so this approach doesn't really help much, and provides no safety checks to ensure you've overloaded for every single `Expression` inheritor.

Implementing an Additional Visitor

So, what is the advantage of the double dispatch approach? The advantage is you have to implement the `Accept()` member through the hierarchy *just once*. You'll never have to touch a member of the hierarchy again. For example: suppose you now want to have a way of evaluating the result of the expression. This is easy…

```
public class ExpressionCalculator : IExpressionVisitor
{
  public double Result;

  public void Visit(DoubleExpression de)
  {
    Result = de.Value;
  }

  public void Visit(AdditionExpression ae)
  {
    // in a moment!
  }

  // maybe, what you really want is double Visit(...)
}
```

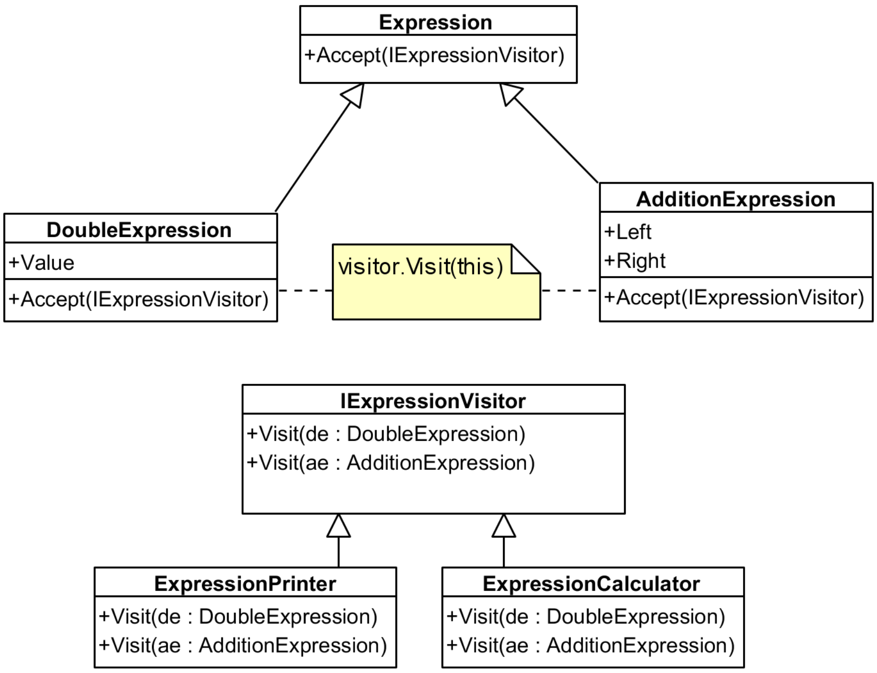
…but one needs to keep in mind that `Visit()` is currently declared as a `void` method, so the implementation for an `AdditionExpression` might look a little bit weird:

```
public void Visit(AdditionExpression ae)
{
  ae.Left.Accept(this);
  var a = Result;
  ae.Right.Accept(this);
  var b = Result;
  Result = a + b;
}
```

The above is a byproduct of an inability to `return` from `Accept()`, so we cache the results in variables `a` and `b` and then return their sum. It works just fine:

```
var calc = new ExpressionCalculator();
calc.Visit(e);
WriteLine($"{ep} = {calc.Result}");
// prints "(1+(2+3)) = 6"
```

The interesting thing about this approach is that you can now write new visitors, in separate classes, even if you don't have access to the source code of the hierarchy itself. This lets you stay true to both SRP and OCP, in addition to making your code a lot easier to understand.


Expression
+Accept(IExpressionVisitor)
DoubleExpression
+Value
+Accept(IExpressionVisitor)
visitor.Visit(this)
AdditionExpression
+Left
+Right
+Accept(IExpressionVisitor)
IExpressionVisitor
+Visit(de : DoubleExpression)
+Visit(ae : AdditionExpression)
ExpressionPrinter
+Visit(de : DoubleExpression)
+Visit(ae : AdditionExpression)
ExpressionCalculator
+Visit(de : DoubleExpression)
+Visit(ae : AdditionExpression)

Acyclic Visitor

Now is a good time to mention that there are actually two strains, if you will, of the Visitor design pattern. They are

- **Cyclic Visitor**, which is based on function overloading. Due to the cyclic dependency between the hierarchy (which must be aware of the visitor's type)

  and the visitor (which must be aware of *every* class in the hierarchy), the use of the approach is limited to stable hierarchies that are infrequently updated.
- **Acyclic Visitor**, which is also based on type casting. The advantage here is the absence of limitations on visited hierarchies but, as you may have guessed, there are performance implications.

The first step in the implementation of the Acyclic Visitor is the actual visitor interface. Instead of defining a `Visit()` overload for every single type in the hierarchy, we make things as generic as possible:

```
public interface IVisitor<TVisitable>
{
  void Visit(TVisitable obj);
}
```

We need each element in our domain model to be able to accept such a visitor but, since every specialization is unique, what we do is introduce a *marker interface* – an empty interface with absolutely nothing in it.

```
public interface IVisitor {} // marker interface
```

The above interface has no members, but we *will* use it as an argument to an `Accept()` method in whichever object we want to actually visit. Now, what we can do is redefine our `Expression` class from before as follows:

```
public abstract class Expression
{
  public virtual void Accept(IVisitor visitor)
  {
    if (visitor is IVisitor<Expression> typed)
      typed.Visit(this);
  }
}
```

So here's how the new `Accept()` method works: we take an `IVisitor` but then try to cast it to an `IVisitor<T>` where `T` is the type we're currently in. If the cast succeeds, the visitor in question knows how to visit our type, and so we call its

`Visit()` method. If it fails, it's a no-op. It is *critical* to understand why `typed` itself does not have a `Visit()` that we could call on it. If it did, it would require an overload for every single type that would be interested in calling it, which is precisely what introduces a cyclic dependency.

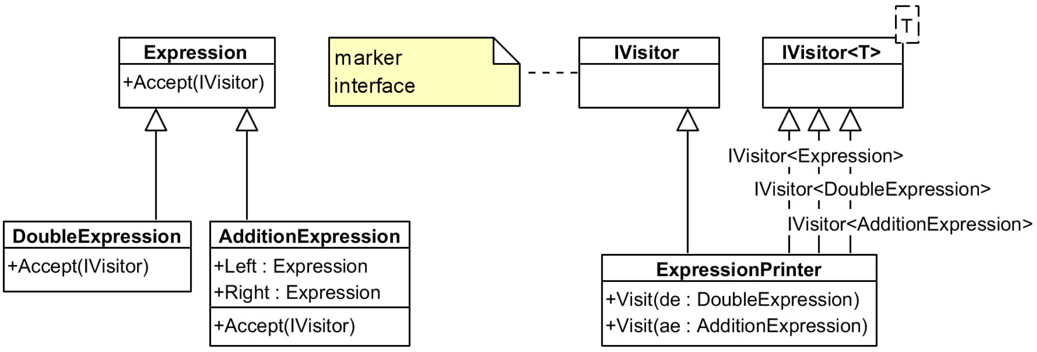
After implementing `Accept()` in other parts of our model (once again, the implementation in each `Expression` class is identical), we can put everything together by once again defining an `ExpressionPrinter`, but this time round, it would look as follows:

```
public class ExpressionPrinter : IVisitor,
    IVisitor<Expression>,
    IVisitor<DoubleExpression>,
    IVisitor<AdditionExpression>
{
  StringBuilder sb = new StringBuilder();

  public void Visit(DoubleExpression de) { ... }

  public void Visit(AdditionExpression ae) { ...  }

  public void Visit(Expression obj)
  {
    // default handler?
  }

  public override string ToString() => sb.ToString();
}
```

As you can see, we implement the `IVisitor` marker interface as well as a `Visitor<T>` for every `T` that we like to visit. If we omit a particular type `T` (for example, suppose I comment out `Visitor<DoubleExpression>`), the program will still compile, and the corresponding `Accept()` call, if it comes, will simply execute as a no-op.

In the above, the implementations of the `Visit()` methods are identical to what we had in the Classic Visitor implementation, and so are the results.

There is, however, a fundamental difference between this example and the classic visitor. The classic visitor used an interface, whereas our acyclic visitor has an abstract class as the root of the hierarchy. What does this mean? Well, an abstract class can have an implementation that can be used as a 'fallback', which is why in the definition of `ExpressionPrinter`, I can implement `IVisitor<Expression>` and provide a `Visit(Expression obj)` method that can be used to handle a missing `Visit()` overload. For example, you could add logging here, or throw an exception.



Functional Visitor

We have already looked at a functional approach to implementing a Visitor when we discussed the Interpreter design pattern, so I won't repeat it here. The general approach boils down to a traversal of a recursive discriminated union using pattern matching on types together with other useful features such as list comprehensions (assuming you're using lists, of course).

The Visitor in a functional paradigm is fundamentally different to the paradigm in OOP. In object-oriented programming, a Visitor is a mechanism for giving a set of related classes additional functionality 'on the side', while ideally being able to

- Group the functionality together
- Avoid type checks, instead relying on interfaces

Pattern matching on a discriminated union is the equivalent of using the `is` C# keyword (`isinst` IL instruction) to check each type. Unlike C#, however, F# will tell you about the missing cases, so it offers greater compile-time safety.

Thus, compared to the OOP implementation, the canonical F# visitor would implement a *reflective visitor* approach.

The implementation of Visitor in F# is problematic for many reasons. First, as we've mentioned before, discriminated unions themselves break the OCP since there's no way to extend them other than to change their definitions. But the Visitor implementation compounds the problem further: since our functional Visitor is essentially a huge `switch` statement, the only way to add support for a particular type is to violate OCP in the Visitor, too!

Summary

The Visitor design pattern allows us to add some distinct behavior to every element in a hierarchy of objects. The approaches we have seen include:

- *Intrusive*: adding a method to every single object in the hierarchy. Possible (assuming you have access to source code) but breaks OCP.
- *Reflective*: adding a separate visitor that requires no changes to the objects; uses `is/as` whenever runtime dispatch is needed.
- *Dynamic*: forces runtime dispatch via the DLR by casting hierarchy objects to `dynamic`. This gives the nicest interface at very large computational cost.
- *Classic* (double dispatch): the entire hierarchy *does* get modified, but just once and in a very generic way. Each element of the hierarchy learns to `Accept()` a visitor. We then subclass the visitor to enhance the hierarchy's functionality in all sorts of directions.
- *Acyclic*: just like the Reflective variety, it performs casting in order to dispatch correctly. However, it breaks the circular dependency between visitor and visitee and allows for more flexible composition of visitors.

The Visitor appears quite often in tandem with the Interpreter pattern: having interpreted some textual input and transformed it into object-oriented structures, we need to, for example, render the abstract syntax tree in a particular way. Visitor helps propagate a `StringBuilder` (or similar accumulator object) throughout the entire hierarchy and collate the data together.