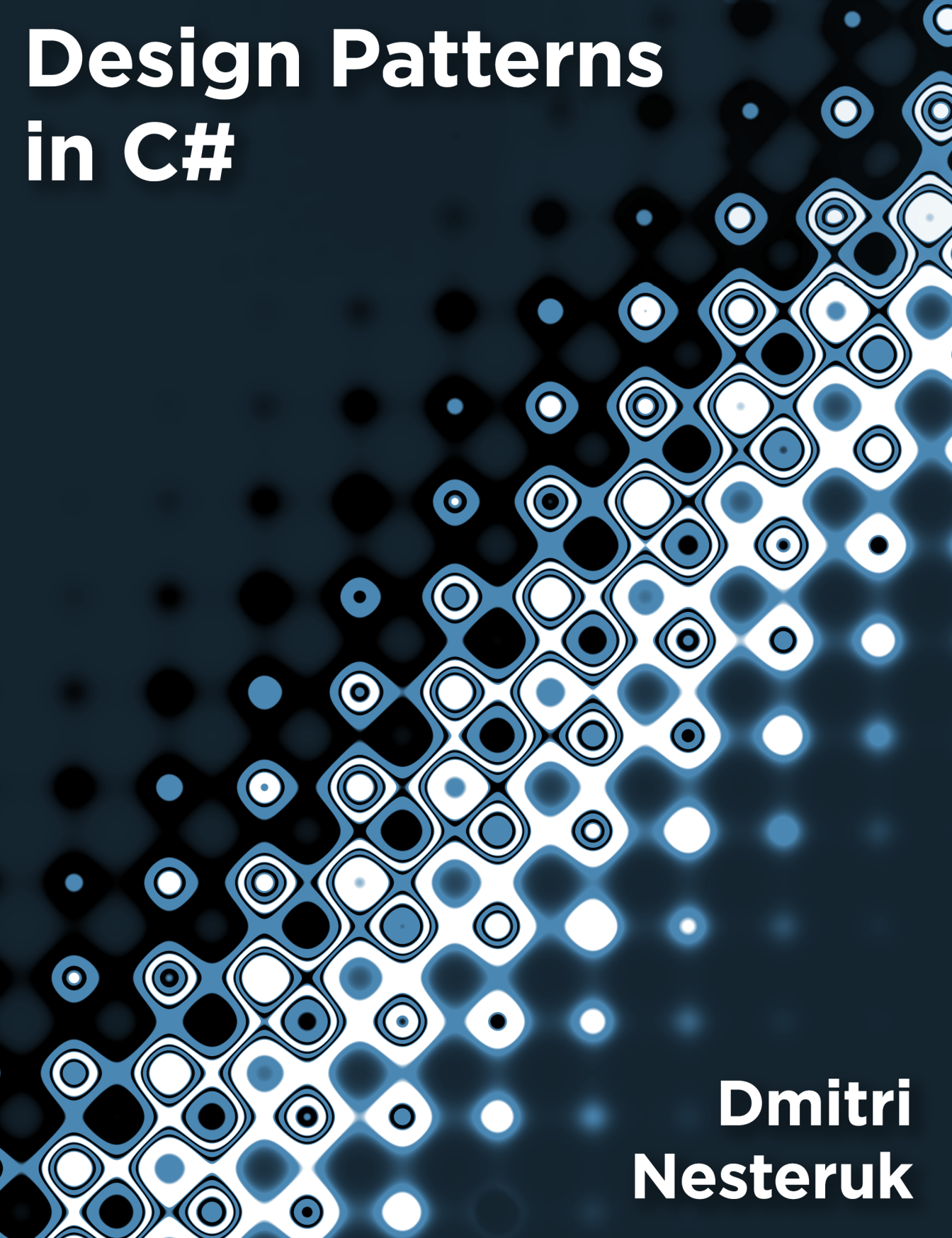
Design Patterns in C#



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

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

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.

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.

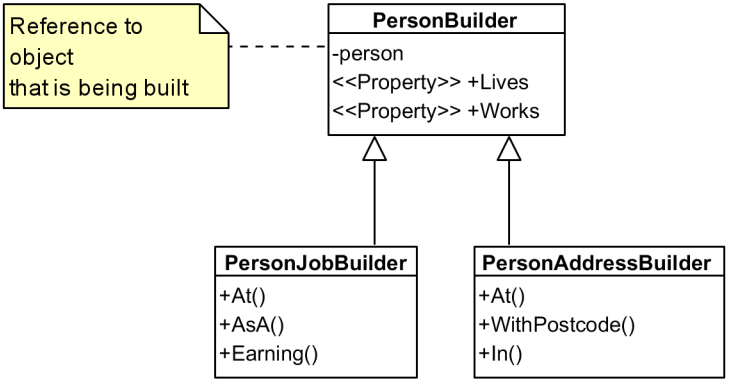
And now, the moment you've been waiting for – an example of these builders in

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#[3]. 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

[3]The Curiously Recurring Template Pattern (CRTP) is a popular C++ technique that allows you to inherit from a template (generic) parameter – something that is sadly 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: `PersonJobBuilder` 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 `Actions` 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.[4]

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.

[4]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.